IX. Evidence Appendix: Copies of Evidence Relied Upon by Appellant with

Exhibit A

JAIN, Alok, "Formal Hardware Verification by Symbolic Trajectory Evaluation," *Carnegie Mellon University Ph.D. dissertation*, July 1997.

The above cited reference was submitted in and Information Disclosure Statement on May 3, 2002 and considered by the examiner on February 29, 2004.

CARNEGIE MELLON UNIVERSITY

FORMAL HARDWARE VERIFICATION BY SYMBOLIC TRAJECTORY EVALUATION

A DISSERTATION
SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY
in
ELECTRICAL AND COMPUTER ENGINEERING

by

ALOK JAIN

Pittsburgh, Pennsylvania
July, 1997

THIS PAGE BLANK (USPTO)

Abstract

Formal verification uses a set of languages, tools, and techniques to mathematically reason about the correctness of a hardware system. The form of mathematical reasoning is dependent upon the hardware system. This thesis concentrates on hardware systems that have a simple deterministic high-level specification but have implementations that exhibit highly nondeterministic behaviors. A typical example of such hardware systems are processors. At the high level, the sequencing model inherent in processors is the sequential execution model. The underlying implementation, however, uses features such as nondeterministic interface protocols, instruction pipelines, and multiple instruction issue which leads to nondeterministic behaviors.

The goal is to develop a methodology with which a designer can show that a circuit fulfills the abstract specification of the desired system behavior. The abstract specification describes the high-level behavior of the system independent of any timing or implementation details. The natural specification of a processor is the instruction set architecture. The specification is defined as a set of abstract assertions defining the effect of each operation on the user-visible state. An implementation mapping is used to relate abstract states to detailed circuit states. The mapping captures the micro-architecture of an implementation of the processor. Symbolic Trajectory Evaluation is used to verify that the circuit fulfills each individual abstract assertion under the implementation mapping. Symbolic Trajectory Evaluation can be considered to be a hybrid approach based on symbolic simulation and model checking algorithms.

The methodology has been applied to the fixed point unit of a superscalar processor that implements the PowerPC architecture. The processor represents a significant leap of complexity compared to previous attempts at formal verification of processors. Our approach seems to be the first one that can truly deal with the complexity of pipeline interlocks.

THIS PAGE BLANK (USPTO)

Acknowledgments

Several people have contributed towards this work. First and foremost among these is my advisor, Randy Bryant. Randy pushed me to the limits of my capabilities and beyond. A large component of this thesis is due to his insistence on the development of methodology, theory and tools that could be applied to some real-world problems (Needless to say that same insistence is also responsible for my rather extended stay at CMU!). The other members in my committee, Jerry Burch, Ed Clarke, Dan Siewiorek and Don Thomas are responsible for steering me in the "right" direction and providing constructive criticisms and suggestions. I would also like to thank Gary York for offering useful feedback during the thesis proposal.

I would like to offer a special thanks to Derek Beatty and Kyle Nelson. Derek laid down the foundation of a methodology for verification of processors. I am thankful to him for several discussions that provided the starting point for this work. Kyle used the methodology and tools to verify parts of the Cobra-Lite processor that provided some concrete results for this thesis. Any robustness in the tools is largely due to his rather irrational and stubborn attitude to use the tools his way rather than my way! I would also like to thank Pankaj Agarwal, Sari Coumeri, Shipra Panda, Vishnu Patankar, Yashodhara Pawar, Laureen Treacy, and Miroslav Velev for proofreading parts of the thesis, sometimes at a moment's notice.

This work would not have been possible without the help of numerous friends over the years. Although I am not listing each of you individually, a big thanks to all of you for an exciting and memorable stay in Pittsburgh and in general being there through the good times and the bad times.

The department has provided a rich and stimulating environment for research. I would like to thank Judy Bandola, Elaine Lawrence, Roxann Martin, and Lynn Philibin for all their advice and help over the past few years. I am also grateful to the Semiconductor Research Consortium for sponsoring and enthusiastically supporting this research.

Finally, I am deeply indebted to my parents for providing this opportunity and offering constant support and encouragement.

THIS PAGE BLANK (USPTO)

Table of Contents

THIS PAGE BLANK (USPTO)

List of Figures

THIS PAGE BLANK (USPTO)

Chapter 1

Introduction

The complexity of hardware systems has been increasing at a rapid rate. The introduction of Very Large Scale Integration (VLSI) techniques has enabled the fabrication of a significantly larger number of transistors on a single chip. A measure of this complexity is the transistor count for single chip processors. Processors have evolved from 10^4 transistors per chip in 1970 to a projected 10^8 transistors per chip in the year 2000. In an effort to increase performance, modern processors use design techniques such as pipelining and parallelism. The resulting interaction between operations and the contention for resources creates the potential for serious design errors. Large design teams and critical time to market requirements are two additional factors that complicate the design process and introduce "bugs" in hardware systems.

Simulation is widely used in the industry to validate hardware system designs. However, to exercise the entire design, an exhaustive simulation requires far too many simulation patterns. This point can be made somewhat dramatically by considering the case of a 256-bit RAM circuit. The RAM circuit has to be validated for 10^{80} possible combinations of the initial states and inputs. If all the matter in the galaxy (10^{17} kg) was used to build computers, and if each computer was the size of a single electron (10^{-30} kg), and if each computer simulates 10^{12} cases per second, and if we began simulation at the time of the big bang (10^{10} years ago), then by now we would be just 0.05% of the way through[1]! The industry, therefore, has to rely on simulating a limited number of simulation patterns which typically exercise a small fraction of the circuit. This opens the door to undetected bugs that could prove to be very costly to the industry. The most recent example is the floating point division bug in the Pentium processor[67]. The bug was not detected inspite of 10^{12} simulation patterns. The bug cost Intel Corporation around $470 million!

1. This cosmic analysis of exhaustive simulation time courtesy of Randy Bryant!

An alternative approach that can overcome the weaknesses of simulation is formal verification. Formal verification uses a set of languages, tools, and techniques to mathematically reason about the hardware system. The form of mathematical reasoning that can be used to verify a circuit is, to a large extent, dependent upon the hardware system. The complexity and vast variety of systems has required researchers to develop and explore several different techniques for formal hardware verification.

This thesis concentrates on hardware systems that have a simple deterministic high-level specification but have implementations which exhibit highly nondeterministic behaviors. Systems that operate on an externally visible stored state such as, memories, data paths, and processors often exhibit these behaviors. The implementation of these systems frequently overlaps the execution of tasks in an effort to enhance performance while maintaining the appearance of sequential execution. At the high level, the sequencing model inherent in processors is the sequential execution model. However, the underlying implementations of these processors use features such as pipelines and multiple instruction issue, which leads to nondeterminism in the implementation. Such implementations contain many subtle features with the potential for serious design errors.

Processors are often implemented as a set of interconnected subsystems. The subsystems have complex interfaces and use nondeterministic protocols to interact with each other. Formal verification of such subsystems presents a unique set of challenges. The goal is to verify the implementation of a subsystem against the more natural high-level specification of the entire system. The verification methodology has to incorporate the ability of defining the environment around the subsystem. The environment defines the set of restrictions and requirements placed on the subsystem by the rest of the system. The restrictions and requirements are in the form of a set of nondeterministic protocols defined on the interface signals. In addition to defining these interfaces, the verification methodology has to account for complex features such as instruction pipelines, pipeline interlocks, multiple instruction issue, multiple cycle instructions, branch prediction, and speculative execution.

This thesis outlines a methodology for formal verification of systems that have a simple and deterministic high-level specification but have implementations that exhibit highly nondeterministic

behaviors[21]. The methodology is able to bridge the wide gap between the high-level specification and the implementation's often radical deviation from the sequential execution model. A methodology for formal verification must ensure that an implementation functions correctly under all possible execution sequences. Since there are an infinite number of execution sequences, we verify each operation individually and then reason about stitching arbitrary operations together to form execution sequences.

Our methodology uses a technique called *Symbolic Trajectory Evaluation (STE)* to perform the verification task. STE was introduced in the past as a modified form of symbolic simulation to perform formal hardware verification[12]. We have extended STE to handle some forms of nondeterminism. STE can be considered to be a hybrid approach that combines the circuit modeling techniques of symbolic simulation with some of the analytic methods found in model checking algorithms.

Our long-term objective is to use our methodology to verify the *Cobra-Lite* processor[65]. The Cobra-Lite is a superscalar processor that implements the PowerPC architecture[63]. The processor has several complex features such as pipeline interlocks, multiple instruction issue, and out-of-order execution. The Cobra-Lite processor represents a significant leap of complexity from all previous attempts at formal verification of processors.

1.1 Our Methodology

The goal is to develop a methodology with which a designer can show that a circuit correctly fulfills an abstract specification of the desired system behavior. The abstract specification describes the high-level behavior of the system independent of any timing or implementation details. As an example, the natural specification of a processor is the instruction set architecture. The specification is a set of *abstract assertions* defining the effect of each operation on the user-visible state. The verification process must bridge a wide gap between the detailed circuit and the abstract specification. In spanning this gap, the verifier must account for issues such as system clocking, pipelines, and interfaces with other subsystems. To bridge this gap between the abstract behavior and the circuit, the verification process requires some additional mapping information. The mapping defines how the abstract state is realized in the detailed circuit. The mapping has both spatial and

temporal information. As an example, consider an abstract model of a fixed point unit (FXU) that receives an operand A from an abstract model of the load store unit (LSU). At the circuit level, the FXU and LSU use a handshaking protocol with the ready (rdyA) and acknowledge (ackA) signals as shown in Figure 1.1. The mapping has to map the abstract signal A into a protocol defined on the datA, rdyA, and ackA signals. The same mapping can be used to verify both the FXU and LSU subsystems.

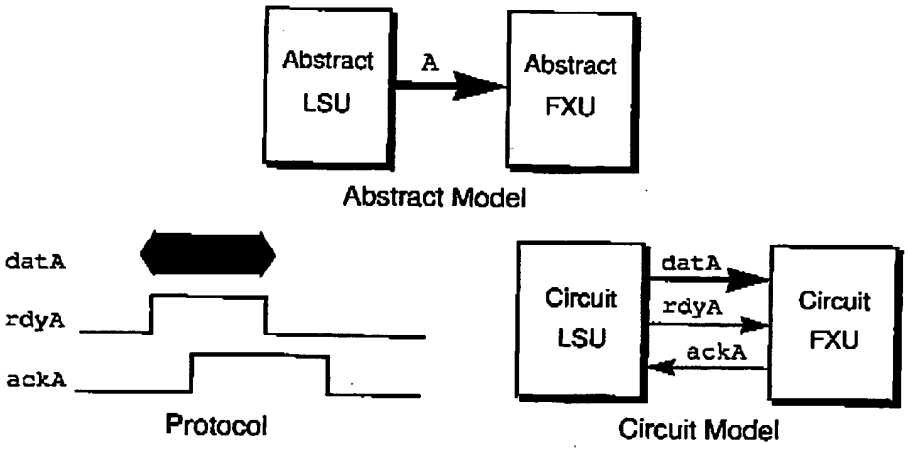


Figure 1.1: Mapping abstract signal into an interface protocol.

Our specification is thus divided into two components: the *abstract specification* and the *implementation mapping*. The distinction serves several purposes. Several feasible circuit designs can be verified against a single abstract specification. An abstract specification can be used to verify multiple implementations of the same architecture. The abstract specification describes the instruction set architecture independent of any pipeline details. The task of the mapping is to relate the abstract specification to the complex temporal behavior and nondeterministic interactions of the pipelined implementation. As an example, an instruction might stall in a pipeline stage waiting to obtain the necessary resources. The order and timing in which these resources are granted often vary, leading to nondeterministic behavior. Our methodology will verify the circuit under all possible orders and timing.

The distinction between the abstract specification and implementation mapping enables hierarchical verification. This thesis concentrates on a single level of mapping that maps the abstract specification into a specific implementation. In the future, one can envision an entire series of

implementation mappings. Each level in the mapping serves to make the assertion more concrete. A verification task could be performed at each level. A series of mappings could also be used to perform modular simulation. Simulation models could be developed at each abstraction level and models at different levels of abstraction could be intermixed using the mapping information.

The goal is to verify that the circuit correctly fulfills the abstract specification. In some sense, the mapping merely serves as hints to guide the verification task. The abstract specification and the implementation mapping are used to generate the *trajectory specification*. The trajectory specification consists of a set of *trajectory assertions*. Each abstract assertion gets mapped into a trajectory assertion. Symbolic Trajectory Evaluation is used to verify the set of trajectory assertions on the circuit. We use the terms trajectory specification and trajectory assertions partly for historical reasons. Our trajectory assertions are a generalization of the trajectory assertions introduced by Seger and Bryant[18]. The justification is that the assertions define a set of trajectories in the circuit. Informally, a trajectory is a sequence of states that represents an acceptable behavior of the circuit.

Once the abstract assertions have been individually verified, the methodology must be able to stitch operations together in order to reason about execution sequences. The mapping has information about how to stitch operations together. Since the abstract assertions use a sequential execution model, stitching operations at the abstract specification level requires a simple sequencing of assertions. The underlying nondeterministic implementation, however, requires operations to interact and overlap in time. The mapping allows the user to describe the interactions and overlap between these operations.

1.2 Specific Details of Our Approach

Figure 1.2 shows the main components of our strategy for formal verification using Symbolic Trajectory Evaluation. We have defined languages to describe the abstract specification and the mapping information. The user provides the abstract specification and the implementation mapping. The abstract specification is defined as a set of declarative abstract assertions. The mapping language is used to define the relation between the assertions and the circuit. In doing so, it defines the interface protocols, the pipeline behavior, and the timing for the circuit. The *Trajectory Generator* takes these two descriptions and generates the trajectory specification. The trajectory specification

is a set of trajectory assertions. The *Symbolic Trajectory Evaluator* is used to verify the set of trajectory assertions on the circuit. The evaluator can accept gate-level circuit models expressed in a subset of Verilog[78] or VHDL[80] languages. We can also verify switch-level circuits[87]. A tool called *Tranalyze*[88] can be used to generate an equivalent gate-level representation for a switch-level circuit.

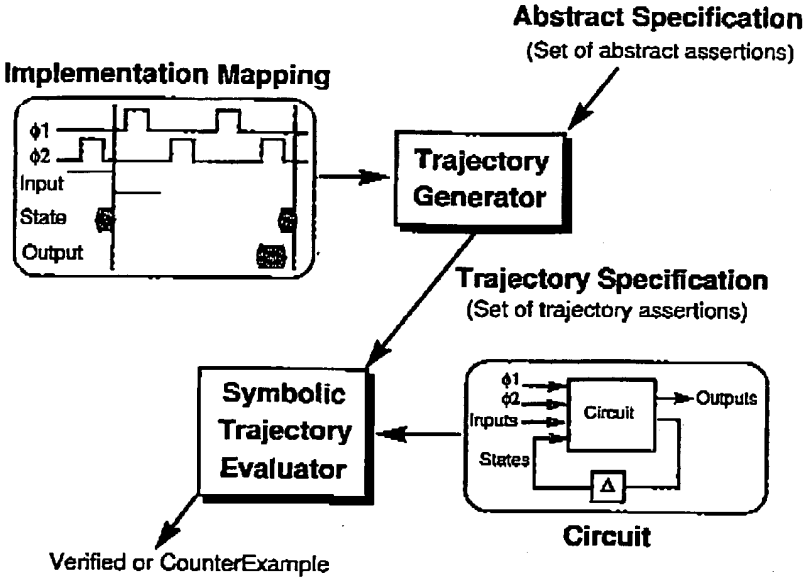


Figure 1.2: Formal verification using Symbolic Trajectory Evaluation.

The Symbolic Trajectory Evaluator verifies that each individual trajectory assertion holds for the circuit. This implies that each individual abstract assertion holds for the circuit under the implementation mapping provided by the user. We, however, want to make the claim that the entire abstract specification holds for the circuit under the mapping. The theoretical foundation behind our approach supports the claim that verifying each individual abstract assertion amounts to verifying the entire abstract specification. And finally, we have to develop a notion of stitching assertions together to reason about execution sequences.

1.3 Application of Our Methodology

Our long-term objective is to develop a methodology capable of verifying modern industrial processors. The touchstone of this work will be the verification of parts of the Cobra-Lite processor. Cobra-Lite is a superscalar implementation of the PowerPC architecture[63] used in IBM's AS/400 Advanced 36 Computer[64][65]. Cobra-Lite is an early version of the Cobra processor with the processor, cache, and I/O interface on a single chip. It is called Cobra-Lite since it is missing about 17 instructions from the required 64-bit PowerPC set. These instructions are primarily floating point instructions that were implemented in software.

There are a number of reasons to pick the Cobra-Lite processor. The Cobra-Lite processor has many complicated features found in modern processors such as forwarding logic, instruction pipelines, pipeline interlocks, multiple cycle instructions, multiple instruction issue, conditionally issued instructions, and out-of-order execution. Yet the design of the processor is not as aggressive as some of the state of the art processors that would completely overwhelm the formal verification task. A more practical consideration is that the verification of the processor is being done at IBM Rochester[1], where both documentation and designers are available to provide a detailed description of the design.

A step towards our long-term objective is to verify individual functional units in the Cobra-Lite processor. This thesis describes the verification of the fixed point unit (FXU). The goal is to verify the FXU against the instruction set specification of the Cobra-Lite processor. Consider an abstract FXU that takes in a source operand S and generates the target T, as shown in Figure 1.3. At the circuit level, the FXU interacts with the load store unit (LSU) for the operands. The FXU and LSU use a protocol on the interface signals. Our goal is to verify the circuit-level view of the FXU against the abstract FXU, replacing the LSU with an environment model that defines the restrictions and requirements on the interface signals.

1. Kyle Nelson, at IBM Rochester, is leading the effort to use our methodology to verify the Cobra-Lite processor. The verification of the Cobra-Lite processor is part of Kyle's PhD work in the Electrical and Computer Engineering Department at Carnegie Mellon University.

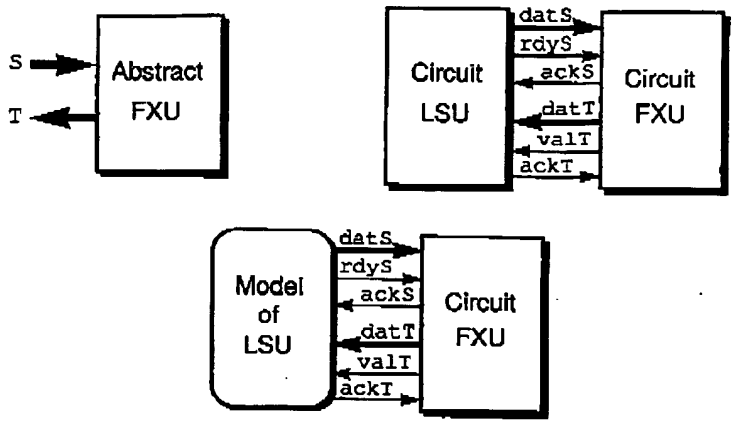


Figure 1.3: Subsystem verification.

1.4 Related Work

There are three main aspects of this thesis: **1.** A methodology for formal hardware verification. **2.** Use of Symbolic Trajectory Evaluation as a verification technique. **3.** Applying the methodology to verify the PowerPC processor. The next few sections discuss some of the related work in each of these areas. More detailed past work has been incorporated in the relevant chapters. A more comprehensive survey of formal verification techniques is available in [1][2]. A summary of past work and a detailed comparison with our approach will be presented later in Section 1.4.9.

1.4.1 Foundation for Our Methodology

Beatty laid down the foundation for our methodology for formal verification of processors[13][15]. The instruction set was specified as a set of declarative abstract assertions. A downward implementation mapping was used to map discrete transitions in the abstract specification into overlapping intervals for the circuit. The overlapping was specified by a *nextmarker*, which defined the nominal end of the current instruction and start of the next instruction. Beatty reduced the task of verification for all possible execution sequences into a check for three separate properties: *Obedience*, *Conformity*, and *Distinction*. The Obedience property is the check to verify that individual abstract assertions hold for the circuit under the implementation mapping. The Conformity check requires that for every specification input sequence, there should be a corresponding

circuit input sequence. The Distinction property requires that two distinct output specification sequences should have distinct output circuit sequences. Beatty's work, however, had one basic limitation. The verification methodology could handle only bounded single behavior sequences. The mapping language was formulated in terms of a formalism called *marked strings*[13]. Marked strings could not express divergent or unbounded behavior.

We have extended the verification methodology to handle a greater level of nondeterministic behavior. As a motivating example, consider a fixed point unit of a processor performing a bitwise-OR operation that fetches two source operands, A and B, from a dual ported load store unit. Assume that the fixed point unit and the load store unit use a handshaking protocol as shown in Figure 1.1. The fixed point unit may have to wait for an arbitrary number of cycles before either of the operands are available. Furthermore, the operands might be received in different orders: Operand A might arrive before B, operand B might arrive before A, or both might arrive simultaneously. The verification methodology should be able to verify the correctness of the circuit under any number of wait cycles and all possible arrival orders. Marked strings cannot express such an operation. Our formulation is in terms of state diagrams that allow users to define unbounded and divergent behavior.

This thesis concentrates mainly on the Obedience property, i.e., to verify that the abstract specification holds for the circuit under some implementation mapping. The conformity and distinction properties are discussed as future work.

Beatty used his verification methodology based on Symbolic Trajectory Evaluation to verify the Hector microprocessor[13][15]. Hector is a simple non-pipelined processor similar to the PDP-11[59][62]. A switch-level implementation of the processor was verified against its instruction set architecture.

1.4.2 Model Checking

Our work has some resemblance to the capabilities provided by model checking tools such as the Symbolic Model Verifier (SMV). SMV is a tool for building a finite model of a system and checking that a desired property holds in that model[29][30][31][33]. The SMV input language can be

used to define a system as a set of interacting finite state machines. The properties are specified in a temporal logic called Computation Tree Logic (CTL).

SMV requires a closed system. The environment is modeled as a set of machines. The state diagrams in our mapping allow the user to define an environment around the system. The state diagrams corresponding to the inputs can be viewed as generators that generate low-level signals required for the operation of the processor. State diagrams corresponding to outputs can be viewed as acceptors that recognize low-level signals on the outputs of the processor. However, there is one essential difference. Though SMV does provide the capability of describing the environment, it does not provide a methodology for rigorously defining these machines and stitching them together to reason about infinite execution sequences. The other difference is that the model checking algorithm in SMV requires the complete next-state relation. It would be impossible to obtain the entire next-state relation for a complex processor. We, on the other hand, use STE to evaluate the next-state function on-the-fly and only for that part of the processor that is required by the specification.

Clarke and others used SMV to verify the cache coherence protocol described in the IEEE Futurebus+ Standard 896.1-1991[34]. The SMV input language was used to create a model of the protocol. Model checking was used to verify that the model satisfied a formal specification of cache coherence describe in CTL. Dill and others used the Murphi verification system to verify the cache coherence protocol of the Scalable Coherent Interface, IEEE Standard 1596-1992[32]. A model of the protocol was built based on the C code that is given as a definition of the SCI standard. Model checking was used to verify that the model satisfied a set of properties required for cache coherence.

1.4.3 Language Containment

Kurshan uses language containment to perform formal verification[27][28]. The specification is represented by a modified form of a deterministic finite state automata called an *L-automata*. The implementation is modeled by a modified form of a nondeterministic finite state machine called an *L-process*[26]. Kurshan refers to the specification as a *task* and to the implementation as a *design*.

Verification is cast in terms of testing whether the formal language $\mathcal{L}(D)$ of a design D is contained in the formal language $\mathcal{L}(T)$ of a task T, i.e., $\mathcal{L}(D) \subseteq \mathcal{L}(T)$.

Kurshan uses several techniques to reduce the complexity of the verification problem. The task T can be broken into a set of subtasks, where each subtask is represented as a deterministic L-automaton. Let i index the set of subtasks. For L-automata, it can be shown that $\mathcal{L}(T) = \bigcap_i \mathcal{L}(T_i)$. Verifying containment on each individual subtask $\mathcal{L}(D) \subseteq \mathcal{L}(T_i)$ implies containment on the entire task $\mathcal{L}(D) \subseteq \mathcal{L}(T)$. This is similar to our approach. In some sense, the task T corresponds to our abstract specification and the individual subtasks T_i correspond to individual abstract assertions.

Kurshan also uses reduction transformations as a basis for complexity management. The reduction transformation is a homomorphism that is defined relative to a task T. The reduction homomorphism can be used to reduce a design D into a simpler D^R, and the task T into a corresponding T^R. Then the containment on the reduced forms $\mathcal{L}(D^R) \subseteq \mathcal{L}(T^R)$ implies containment on the more complex forms $\mathcal{L}(D) \subseteq \mathcal{L}(T)$. The reduction eliminates from the design extraneous or redundant details that are not relevant to the task T. In our methodology, Symbolic Trajectory Evaluation exercises only those parts of the circuit that are relevant to the abstract assertion.

The reduction transformation can also provide a basis for hierarchical verification. The inverse of the reduction serves as a *refinement transformation*. Assume that D^R and D are the designs at two different levels of abstraction hierarchy, where D^R is the design at the abstract level and D is the design at the more concrete level. Then a task verified at the more abstract level would remain valid for the more concrete level. This thesis concentrates on a single level of the implementation mapping. In the future, one can envision an entire series of mappings to provide a basis for hierarchical verification. In that case, our implementation mappings would serve the same role as Kurshan's refinement transformations.

Kurshan's verification methodology has been incorporated into a tool called COSPAN. The tool has been used to formally develop and verify the layered design of communications protocols and their implementation in hardware used by several development projects at AT&T and to verify an arbiter circuit through a four-level hierarchy[2][25].

1.4.4 Symbolic Simulation

In the past, ternary simulation has been used to verify circuits[9][10]. Assuming a monotone excitation function, the simulation algorithm can ensure that any binary values resulting from simulating patterns containing X's would also result when the X's are replaced by any combination of 0's and 1's. Thus the number of patterns that must be simulated to verify a circuit can often be reduced dramatically by representing many different operating conditions by patterns containing X's. For example, ternary simulation can verify that a particular sequence of actions will yield a binary value on some node regardless of the initial state by verifying that this value results when starting from an initial state where all nodes are set to X. This requires far less effort than analyzing the effect of the action on all possible initial binary states. Ternary simulation, however, errs on the side of pessimism for the sake of efficiency. The simulation will sometimes produce a value X, even where exhaustive case analysis would indicate that the value should be binary. A reason for this pessimism is that the ternary extension of the Boolean operators do not obey the law of excluded middle, since $X + \overline{X} = X$, where $+$ and $^-$ have been extended into the ternary domain.

Ternary symbolic simulation can be used to overcome some of these limitations[7]. The simulation algorithm is extended to operate not just on scalar values 0, 1, and X, but also on a set of symbolic values. Each symbolic value indicates the value of a signal for many different operating conditions, parameterized in terms of a set of symbolic Boolean variables. A single symbolic simulation sequence captures multiple ternary scalar simulation sequences. Symbolic simulation can be used to resolve the interdependencies among signal values and avoid the pessimism due to the law of excluded middle. The symbolic simulator can compute $a + \bar{a} = 1$, where $+$ and $^-$ have been extended into the symbolic domain. By combining the expressive power of symbolic values with the computational efficiency of ternary values, we can trade off precision for ease of computation.

Researchers have used symbolic simulation as a technique for performing formal hardware verification[3][4][5]. Formal verification based on symbolic simulation can efficiently model circuit behavior with more detailed circuit and timing models. Symbolic simulation is one of the few techniques that can model system behavior at a level of timing granularity finer than complete clock cycles.

Bose and Fisher used symbolic simulation to verify synchronous pipelined circuits[5]. They used an abstraction function to map a pipelined circuit to an abstract nonpipelined version. The abstraction function is represented as a circuit, so that the abstraction function can be evaluated by the symbolic simulator. They, however, require the user to identify each and every storage location in the circuit. They introduced Boolean variables for each input and state variable and computed the next-state function for each state variable. It would be impossible to identify all the storage locations and obtain the next-state functions for each storage location for systems such as complex processors.

1.4.5 Symbolic Trajectory Evaluation

Symbolic Trajectory Evaluation (STE) has been used earlier to verify trajectory assertions. Beatty mapped each abstract assertion into a set of symbolic patterns[15]. STE was used to verify the set of symbolic patterns on the circuit. The set of symbolic patterns corresponded to single sequence of states in a state diagram. Seger and Bryant extended STE to perform fixed point computations to verify a single sequence of states augmented with a limited set of loops[18]. In our work trajectory assertions are general state diagrams. We have extended STE to deal with generalized trajectory assertions.

STE has been used to verify memory arrays such as on-chip caches and register files. Pandey and others used the VOSS STE system to verify a multi-ported register file unit of a PowerPC processor and the tags unit for a data cache circuit from a recent PowerPC design[20]. VOSS is a system for formal hardware verification that provides a functional language interface to Symbolic Trajectory Evaluation[14]. The number of variables required to verify properties of these arrays was approximately logarithmic in the number of memory locations, thus ameliorating the state explosion problem. Pandey and others also used the VOSS system to verify two content addressable memories from a recent PowerPC design, a block address translation unit, and a branch target address cache unit[22]. New encoding techniques were developed to contain the exponential growth in the space requirement with increasing memory sizes. All these circuits were modeled at the switch level, so that the verification was performed on the actual design.

Hazelhurst and Seger have explored a hybrid approach based on theorem-proving and STE[16][17]. STE is used to prove low-level properties of the circuit. A set of inference rules are used to compose the results of STE in a theorem proving environment. The hybrid approach was able to verify a 64-bit multiplier by verifying the individual adders using STE and then composing the results to show that the adders were properly connected[16]. Aagaard and Seger used the hybrid approach to verify a radix-eight, pipelined, IEEE double-precision floating point multiplier[19].

1.4.6 Verification with Uninterpreted Functions

Burch and Dill have recently proposed a verification technique to verify pipelined and superscalar processors[46][53]. The user has to provide a behavioral description of the implementation and specification. The specification describes the instruction set architecture of the processor. The implementation is described at a level of abstraction that exposes relevant design issues such as pipelining. Each of these descriptions is compiled into a transition function through symbolic simulation. Let $\mathcal{F}_{Impl}$ and $\mathcal{F}_{Spec}$ denote the transition functions of the implementation and specification respectively. The correctness criteria is shown in the commutative diagram in Figure 1.4, where $\mathcal{A}$ is an abstraction function. The integer m is needed to keep the specification and implementation "in sync". For example, if the instruction does not fetch an instruction due to a load interlock, then $m = 0$. For a superscalar processor, $m \geq 1$. Burch and Dill avoid the need to explicitly define the abstraction function. Instead, the abstraction function is implicitly defined by flushing the pipeline and then using a projection function to extract only the programmer-visible state, i.e., $\mathcal{A}(q_{impl}) = Proj(\mathcal{F}_{Flush}(q_{impl}))$. The correctness criteria can now be written as $\forall q_{impl}\, \exists m\, [\mathcal{A}(\mathcal{F}_{Impl}(q_{impl})) = \mathcal{F}^{m}_{Spec}(\mathcal{A}(q_{impl}))]$. Burch and Dill use a quantifier-free first-order logic of uninterpreted functions and predicates with equality and propositional connectives. Uninterpreted functions are used to represent functional units. Propositional connectives and equality are used in describing control and comparing the specification and implementation.

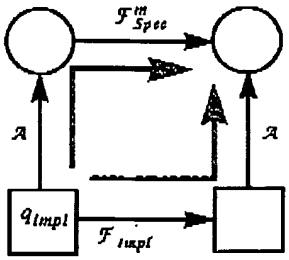


Figure 1.4: Burch and Dill's commutative diagram for correctness criteria.

We, on the other hand, require the user to define an explicit implementation function $\mathcal{J}$. The function $\mathcal{J}$ maps an abstract state into a set of sequences of implementation states. Our correctness criteria is based on a notion of stimulus and response. Our correctness criteria is illustrated in Figure 1.5. The initial abstract state is mapped into a set of sequences that serves as the stimulus for the circuit. The final abstract state is mapped into a set of sequences that defines the desired response from the circuit. The function $\mathcal{F}^{*}_{Impl}$ is required because STE has to be run for as long as specified by the stimulus and response. The implementation function defines a nextmarker which is the timepoint where STE needs to start checking the response. The correctness criteria can be roughly stated as:

$\forall(q_{spec} \in Q_{Spec})\ \forall(\sigma_{impl} \in \mathcal{J}(q_{spec}))\ [\mathcal{F}^{*}_{Impl}(\sigma_{impl}) \subseteq \mathcal{J}(\mathcal{F}_{Spec}(q_{spec}))]$, where σ_{impl} represents a sequence of implementation states and the subset property has been extended to sequences.

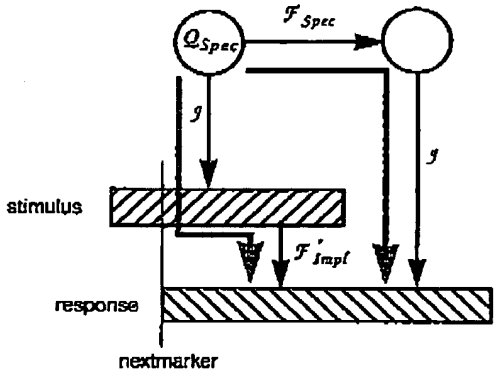


Figure 1.5: A notion of our correctness criteria.

There are several other differences between Burch and Dill's approach and our approach. Burch and Dill's approach requires a dichotomy between the data path and control. They focus on the control assuming that the data path is correct. The user has to provide an abstract description of the implementation. On the other hand, we operate directly on the circuit design. And we use Binary Decision Diagrams[70], which enable us to verify both the data path and control. Furthermore, Burch and Dill require that the specification and implementation have the same set of inputs and no outputs. In other words, the abstraction function is limited to internal states in the processor. In addition to mapping internal states, our implementation mapping allows the user to map an abstract input or output at the specification level into a protocol on multiple signals at the implementation level.

Burch and Dill have used their automated technique to verify a pipelined implementation of a subset of the DLX architecture[46][51]. The DLX architecture was designed by Hennessy and Patterson to teach the basic concepts used in the MIPS 2000 and other RISC processors of that generation[61]. The pipelined implementation was verified against its instruction set architecture. Their implementation had a 5-stage pipeline with a load interlock. The user provided an abstract description of the implementation. Subsequently, Burch used the technique to verify a superscalar implementation of a subset of the DLX processor[53]. To deal with the complexity of the superscalar processor, Burch split the commutative diagram into three separate commutative diagrams[53][54]. Their approach, however, has the limitation that the abstraction function can become very complex in the presence of pipeline interlocks. Burch suggests modifying the processor to add a control input that can be used to force the stalling of an instruction at any stage where the instruction can be stalled in the pipeline. This, however, requires manual intervention to modify the processor and devise a new flushing schedule.

1.4.7 Theorem Proving and Processor Verification

Theorem proving has been used in the past to verify nonpipelined processors[40][41][42][43][44]. Subsequently, there have been a number of attempts to use theorem proving to verify pipelined processors.

Srivas and Bickford used the Clio system to verify a pipelined processor called the Mini Cayuga[45]. Clio is a functional language-based verification system that verifies properties of programs. The programs are written in Caliban, a polymorphic, strongly typed, lazy functional language[37]. The properties to be proved are expressed in the Clio assertion language and proved interactively with the Clio theorem prover. The Mini Cayuga is a scaled down version of a RISC processor, Cayuga, that was designed in a graduate level VLSI course at Cornell University[60]. The Mini Cayuga is a three-stage instruction pipelined processor with a single interrupt. Both the specification and the implementation of the microprocessor were specified in Caliban. The correctness criteria was specified in the Clio assertion language. The correctness criteria was to relate traces at the implementation and specification levels by using an abstraction function provided by the user. The verification assumed that the processor started in a known power up state.

Windley and Coe used the HOL system to verify a pipelined processor called Uinta[48][49]. HOL is a theorem proving system developed at the University of Cambridge that is based on higher order logic[35]. Uinta has a five-stage pipeline with data and control hazards. The processor uses data forwarding and delayed branches to deal with these hazards. Windley developed the generic interpreter theory that provides a standard model for the specification and verification of nonpipelined processors[44]. The generic interpreter theory used two separate abstraction functions, i.e. the temporal abstraction function and the data abstraction function to relate the implementation to the specification. The temporal and data abstractions, however, are not orthogonal for pipelined processors. Windley and Coe used a single abstraction function that intermixed the data and temporal abstractions to verify the Uinta processor. They used case analysis to deal with pipeline stalls and were able to limit the proof to one stall case split by using a lemma to show that the pipeline cannot stall twice in a row.

Researchers have used the Prototype Verification System to verify a number of pipelined processors[47][52][55]. The Prototype Verification System (PVS) is an environment for specification and verification that has been developed at SRI[38]. PVS combines a highly expressive specification language based on higher order logic, with a very effective interactive theorem prover in which most of the low-level proof steps are automated.

Cyrluk and others used PVS to verify the Saxe pipelined processor[47] and Burch and Dill's pipelined DLX processor[55]. The specification and implementation were specified as state transition systems. The correctness criteria related traces at both levels by using an abstraction function provided by the user. The limitation is that the user has to provide the number of cycles that it takes the implementation to complete an instruction as a function of the current state and future instructions. However, for modern processors, the number of cycles for completing an instruction might be indefinite due to pipeline interlocks or stalls.

Srivas and Miller have used PVS to verify the AAMP5 microprocessor[52]. The AAMP5 belongs to a family of Rockwell proprietary microprocessors based on the Collins Adaptive Processor System[58]. The AAMP5 has a large complex instruction set, multiple data types and addressing modes, and a microcoded, pipelined implementation. The implementation of the AAMP5 has some 500,000 transistors. A factor complicating the verification of the AAMP5 is that it can stall for an arbitrary number of cycles due to memory wait states and stack adjustments. Srivas and Miller decomposed the verification problem into three sub-problems: **1.** A part that reasons exclusively about the stalling behavior. **2.** A part that reasons about individual instructions in the absence of stalling. **3.** A part that combines the first two parts. The correctness of pipeline stalling was characterized by a set of formulas called *general verification conditions* since they are common to all instructions. The correctness of individual instructions in the absence of stalling was characterized by a set of *instruction-specific verification conditions*. Srivas and Miller comment on the difficulty of verifying the general verification problems as follows: *"... Since the general verification conditions are not in the form of a property that relates states that are a finite distance apart, their proof is not as automatic as that of the instruction-specific verification conditions. The proof involves induction on the length of time taken by the memory to respond or size of the stack adjustment needed and requires involvement by an expert."* Srivas and Miller needed to formulate only a few general verification conditions since the AAMP5 has only three stalling situations and all of these are instruction independent. On the other hand, Cobra-Lite has much more complex stalling situations that are dependent on the instructions (or at least the instruction formats).

Researchers have used the ACL2 theorem prover to verify pipelined processors[56][57]. ACL2 stands for "A Computational Logic for Applicative Common Lisp." ACL2 is both a mathematical

logic and a set of tools which can be used to construct proofs in the logic[36][39]. The logic is a quantifier-free first-order logic of total recursive functions.

Brock and others used the ACL2 theorem prover to verify the CAP processor[56]. CAP is a DSP co-processor optimized for communications signal processing under development by Motorola[66]. CAP has a three-stage instruction pipeline and does not have any control logic to deal with pipeline hazards. It is the task of the programmer to avoid pipeline hazards in their DSP application code. Brock used the same approach as Burch and Dill, thus avoiding the need to define an explicit abstraction function. ACL2 did not have to deal with the complexity of pipeline interlocks since the verification was performed on only hazard-free code. It should, however, be acknowledged that depending on the verification strategy, compiler assumptions can be more difficult to handle than pipeline interlocks.

Sawada and Hunt used the ACL2 theorem prover to verify an out-of-order pipelined processor[57]. They designed their own processor. The processor includes out-of-order execution and speculative instruction fetch. In order to avoid pipeline hazards, the issuing logic suspends the issue of any instruction which may cause a hazard. Once an instruction is issued, it is guaranteed that no hazard will occur due to the instruction. Sawada and Hunt use the same approach as Burch and Dill, avoiding the need to define an explicit abstraction function. They use a table to store an execution trace of instructions representing states in the implementation. The table representation helps to easily define various pipeline properties, such as the absence of WAW-hazards. In a sense, the table captures the past history of the processor. In Symbolic Trajectory Evaluation, the initial state of the circuit is set to the most general state that reflects all possible past histories for the processor.

1.4.8 Miscellaneous

All of the past work mentioned so far concentrated on comparing the instruction set architecture with an implementation of the processor. Some researchers have used combinational equivalence checking to compare the RTL, gate or transistor level representations of a processor. Appenzeller and Kuehlmann used Verity to verify the PowerPC microprocessor[50]. Verity is a verification tool developed at IBM for functional verification of large transistor and gate-level circuits by using

Binary Decision Diagrams. Appenzeller and Kuehlmann compared an implementation at the register transfer-level with the transistor-level implementation. The user had to identify the storage locations in both representations and Verity was used to perform combinational verification.

1.4.9 Summary of Past Work

Most of the past work in formal verification of pipelined processors has concentrated on relatively simple processors. Some of the limitations of previous work and their relation to our strategy for verifying the Cobra-Lite processor are as follows:

- Most of the past work required the user to provide some abstracted version of the processor implementation. Either the datapath had to be abstracted away or the user had to give a cycle level transition function for the implementation. Symbolic Trajectory Evaluation enables us to verify an actual gate-level circuit design of the Cobra-Lite processor. The use of Binary Decision Diagrams enable us to verify both the data path and control simultaneously.
- Past work has been limited to processors with very simple implementations. The Cobra-Lite processors is a real design that is significantly more complex both in terms of the size and features incorporated in the implementation. It is doubtful whether any of the existing techniques would be able to verify the entire processor as a single entity. The Cobra-Lite processor is designed as a set of interconnected subsystems or functional units. We are advocating decomposing the problem of verification of the entire processor into smaller subproblems of verification of individual functional units. We would like to verify each individual functional unit against the instruction set architecture of the entire processor and then reason about the interactions between various functional units. This thesis is the first step in this direction and concentrates mainly on subsystem verification.
- All the previous verification techniques assumed that the specification and the implementation had the same set of inputs and outputs. The abstraction function related only internal states in the specification and implementation. In other words, the correctness criteria actually compared the pipelined processor with a nonpipelined version of the processor. These techniques are not

suitable for subsystem verification. Our implementation mapping maps abstract inputs and outputs into protocols at the lower level. A major part of our work is to provide a methodology for rigorously defining the environment around the functional unit to be verified.

- Past work has dealt with simple pipeline interlocks. Most of the past work were able to limit the verification to a single cycle pipeline interlock. It is conceivable that the verification could be extended to pipeline interlocks for an indefinite number of cycles. However, the previous approaches have not clearly shown that this can be done without significantly increasing the complexity of the verification task. The functional units in the Cobra-Lite processor have complicated control logic to deal with pipeline interlocks. There are a number of reasons for an instruction to stall for an indefinite number of cycles in a pipeline stage. Our verification has to prove the correctness of the functional units under all possible scenarios and under all possible number of stall cycles.

It is worthwhile to compare various different approaches to processor verification as regards the degree of automation and robustness to minor modifications in the implementation. We are attempting to use automated techniques to verify a real processor. It should, however, be acknowledged that comparisons between automated and manual techniques are very subtle. Our approach is probably more automated than some of the theorem proving efforts which require considerable human guidance to perform the verification. However, our approach requires an up-front manual effort to provide the implementation mapping. The implementation mapping can become quite complex and requires an understanding of several low-level details in the gate-level implementation of the processor. Burch and Dill's approach can be considered to be more automated since they avoid the need to explicitly define an abstraction function. The abstraction function is implicitly defined by flushing the pipeline and then using a projection function to extract only the programmer-visible state. The problem is that they have to modify the processor and devise a new flushing schedule to limit the complexity of the abstraction function.

Another point of comparison is the robustness of different approaches with respect to minor modifications in the implementation. Some theorem proving efforts require the user to define both an abstracted version of the implementation and the abstraction function. These theorem proving efforts are the least robust since they could require redefining both the implementation and the

abstraction function. The comparison between our approach and Burch and Dill's approach is not so clear. Our approach would not require any effort to redefine the implementation, since we operate directly on the gate-level implementation. The implementation mapping, however, might have to be redefined. Burch and Dill do not require the user to provide the abstraction function. The implementation, however, might have to be recompiled into a transition function. Also, the user might have to define a new flushing schedule for the modified processor.

1.5 Limitations of our Work

It is worthwhile to discuss some of the limitations of our approach. Here, we will briefly state the major limitations of our approach. Several techniques to deal with some of these limitations are discussed as future work in Chapter 9.

- Our approach is targeted towards verification of systems that have a simple and deterministic high-level specification but whose implementations can exhibit a variety of possible nondeterministic behaviors. Processors, memories and data paths are an example of such class of systems. However, we cannot verify systems, such as a cache coherence protocol, that do not have a simple high-level specification.
- A major limitation is that Symbolic Trajectory Evaluation requires a deterministic circuit-level model of the implementation. Let us assume that we are attempting to verify a system of multiple interacting finite-state machines. Our approach would require a detailed low-level model of all these machines and verify them as one single entity. It is not possible to verify each individual finite-state machine and then reason about their interaction at some abstract level.
- The user is required to provide the implementation mapping. As stated before, the implementation mapping can become complex and requires a detailed understanding of the circuit.
- Our approach does not scale well for more complex processors. The trajectory generation phase generates all possible nondeterministic behaviors in the subsystem. Increasing the pipeline depth or the number of resources in a subsystem can lead to an exponential blowup in the number of possible nondeterministic behaviors. The methodology has been used to verify the fixed

point unit of the Cobra-Lite processor. The fixed point unit has a shallow three-stage pipeline. Several modifications and enhancements would be required to deal with processors with deeper pipelines and superscalar execution.

- Symbolic Trajectory Evaluation uses Binary Decision Diagrams to verify both the data path and control simultaneously. Our approach, therefore, suffers from the typical limitations associated with Binary Decision Diagrams. Careful attention has to be paid to the variable ordering to limit the size of the Binary Decision Diagrams.

1.6 Overview of the Thesis

The essential ingredients of our formal verification methodology can be described as: **1.** The languages defining the abstract specification and the implementation mapping. **2.** The trajectory generation phase. **3.** The verification algorithm (including Symbolic Trajectory Evaluation). **4.** The theory that puts it all together. The thesis follows this order.

Chapter 2 explores various notations for expressing abstract specifications. Hardware Description Languages are not suitable for *specifying* hardware. This chapter introduces a *Hardware Specification Language* to specify hardware. The basic form of the Hardware Specification Language defines a nondeterministic Moore machine. The transitions in this machine are defined as a set of declarative abstract assertions. The chapter describes symbolic and vector extensions for the language to ease the task of writing specifications. The syntax of the language is shown with examples.

Chapter 3 introduces the form of the implementation mapping. The implementation mapping is defined in terms of a variation of state diagrams called *control graphs*. The basic form of the implementation mapping consists of a *main machine* and a set of *map machines*. The main machine defines the flow of control of individual system operations. The map machines define a spatial and temporal mapping for each abstract state in the specification. The chapter describes symbolic and vector extensions to ease the task of writing the mapping. The syntax of the mapping language is shown with examples.

Chapter 4 describes the trajectory generation phase. The abstract specification and mapping are used to generate the trajectory specification. Each abstract assertion gets mapped into a trajectory assertion. The main step in trajectory generation is the composition of control graphs. This chapter describes the composition operator in detail. The main elements of the trajectory generation phase are illustrated with a simple example.

Chapter 5 describes various verification algorithms that could be used to verify the set of trajectory assertions on the circuit. This chapter introduces the term *Trajectory Checking*. The term trajectory checking has been coined to denote its mixed heritage from model checking and Symbolic Trajectory Evaluation. Trajectory checking uses a relaxation algorithm to label the trajectory assertion with circuit states. There are two forms of trajectory checking, i.e., a set-based and a lattice-based approach. The lattice-based approach can be viewed as an approximation of the set-based approach. Symbolic Trajectory Evaluation can now be viewed as a technique to overcome the limitations of trajectory checking. The trajectory checking algorithm requires the entire next-state function for the circuit. Symbolic Trajectory Evaluation enables the next-state function to be computed on-the-fly and only for that part of the circuit required by the specification. The chapter discusses extensions to Symbolic Trajectory Evaluation to deal with our generalized model of trajectory assertions.

Chapter 6 is where we put it all together. This chapter revisits some of the concepts introduced in earlier chapters and develops a firm theoretical foundation for formal hardware verification using Symbolic Trajectory Evaluation. The previous chapter verified that each individual abstract assertion holds for the circuit under the mapping. The theoretical foundation behind our approach supports the claim that the entire abstract specification holds for the circuit under the mapping.

Chapter 7 describes some preliminary work in reasoning about execution sequences. This chapter introduces the concept of closure construction as a means of stitching operations or assertions together to form infinite execution sequences.

Chapter 8 applies our methodology to verify the fixed point unit of a superscalar implementation of the PowerPC architecture. The chapter gives an overview of the fixed point unit, and describes details of the abstract assertion and the implementation mapping that were used to verify the fixed

point unit. The chapter describes the result of trajectory generation and Symbolic Trajectory Evaluation.

Finally, Chapter 9 has a discussion of future work and conclusions.

Appendices A and B contain the man pages describing the syntax of the Hardware Specification Language and the implementation mapping.

THIS PAGE BLANK (USPTO)

Chapter 2

Abstract Specification

This chapter explores notations for expressing abstract specifications. Any specification notation should be both simple and abstract, providing designers with a natural way to specify hardware. On the other hand, it should allow designers to specify low-level details that are relevant to the design task. Though this thesis concentrates on formal verification, a specification notation has to support other CAD tasks such as simulation and high-level synthesis. A high-level simulation task allows the user to informally check the validity of the specification. A specification can be mapped into concrete circuit realization in a variety of different ways. At one end of the spectrum is a completely manual custom designed circuit. At the other end is a completely automated high-level synthesis task. After obtaining a realization, one has to verify that the circuit fulfills the specification. There are several CAD tasks that can be used to perform this verification. Simulation can be used to establish a degree of confidence in the circuit. Formal verification can be used to mathematically reason about the correctness of a circuit with respect to a specification. A specification notation has to serve the needs of all the above mentioned CAD tasks.

High-level behavioral constructs in Hardware Description Languages (HDLs) seem to be the obvious choice for this notation. Designers are familiar with HDLs and they have been used extensively in the past for simulating and reasoning about high-level descriptions. However, HDLs tend to overspecify systems, since they tend to describe a sample implementation rather than an abstract specification. Hardware Description Languages are useful languages for *describing* hardware. However, they are not suitable for *specifying* hardware. We introduce a *Hardware Specification Language* to specify hardware.

2.1 Hardware Description Languages

Hardware Description Languages have been widely used to describe systems at varying levels of abstraction. Some of the more popular Hardware Description Languages in the market are Ver-

ilog[78], VHDL[80], and UDL/I[79]. The behavioral constructs in these languages can be used for specifications. Typically HDLs also offer structural constructs to describe low-level realizations. Therefore, they provide a single framework for describing both the specification and circuit design. Designers have been using HDLs extensively to model systems. Commercial simulators are available to simulate most HDL descriptions at all degrees of abstraction. In recent years, there have been several attempts to use an HDL subset as the front end for high-level synthesis[81][82][83][84]. Therefore it seems that all that has to be done is to adopt HDLs as a specification notation for formal verification.

The first problem with HDLs such as Verilog and VHDL is that they lack formal semantics. Recently, there have been some attempts to define formal semantics for at least a subset of VHDL[77]. Furthermore, UDL/I has a documented formal semantic[79]. For the moment, let's assume that formal semantics have been defined for these HDLs. Let us adopt Verilog HDL as our specification notation and attempt to specify the high-level behavior of a stack.

Consider a realization of a stack where each element in the stack is moved down on a push operation and each element is moved up on a pop operation. Such a stack is called a *Moving Data Stack*. Let's write a specification for such a stack using Verilog HDL. Such a specification is shown in Figure 2.1. All timing has been abstracted out by assuming a positive edge triggered stack, although this introduced an explicit clock as an implementation artifact. The essential component of the specification is the *always* block. The always block is activated at all positive edges of the clock, causing a push or pop operation to be performed on the control signal op. This example shows several shortcomings of the specification. Note that the iterations (for loops) have to be carefully set so as to shift the bottom elements first during a push operation and the top element first during a pop operation. In a push operation, the data input is stored in the stack only after shifting all elements in the stack. In the pop operation, the output is set before shifting all elements in the stack. In other words sequencing of operations becomes a major issue in these behavioral descriptions[1]. Furthermore note that the hold operation is not explicitly specified. The semantics imposed by a simulator on these languages is that if no action is specified, then the signals retain

1. It should be acknowledged that some of the sequencing issues can be dealt with the use of non-blocking assignments.

their previous values. The semantics implicitly incorporates the hold operation. Note that the semantics impose the same functionality on the illegal (op = 2'b11) operation. In a circuit, the environment around the stack might ensure that op is never set to 2'b11. In that case the illegal operation is a don't care condition for the stack. Imposing the hold functionality to the illegal operation is overspecifying the stack. Finally, let's take a look at the boundary conditions for the stack. According to the behavioral description, a push on a full stack discards the bottom-most element in the stack and inserts the data input into the top of stack. A pop on an empty stack sets the data output to the value stored on the top of stack. The value on the top of stack is dependent on the past history of operations on the stack. If in the past the stack has never been filled to capacity, then this value is the initialized value for the top of stack. However, if in the past the stack has been filled to capacity, then the value is the element that was stored at the bottom of the stack.

```
module MovingDataStackSpec(dataOut, dataIn, op, clock)
    output dataOut;
    input [0:1] op;    // Push=2'b00, Pop=2'b01, Hold=2'b10, Illegal=2'b11
    input dataIn;
    input clock;

    reg[0:1023] Mstack;    // Defining a moving stack of 1024 elements.
    reg dataOut;
    reg i;

    always @(posedge clock)
    begin
        if (op == 2'b00)    // Push Operation.
        begin
            for (i = 1023; i > 0; i = i - 1) Mstack[i] = Mstack[i-1];
            Mstack[0] = dataIn;
        end

        if (op == 2'b01)    // Pop Operation.
        begin
            dataOut = Mstack[0];
            for (i = 0; i < 1023; i = i + 1) Mstack[i] = Mstack[i+1];
        end
    end
endmodule
```

Figure 2.1: A Verilog HDL specification of a moving data stack.

One of the goals in verification is that a single specification should be able to check several possible circuit designs. Towards that end, let us consider another realization of the stack. This realization is a *Stationary Data Stack* based on a RAM and a top of stack pointer. The pointer points to the first free location in the stack. Unfortunately, the moving data stack specification cannot serve

as a specification for the stationary stack. A Verilog HDL behavioral description for a stationary stack is shown in Figure 2.2. The top of stack `tos` is a 10-bit register, which can store integers in the range 0 to 1023. Note that `tos` points to location 1023 when the stack has 1023 elements. A subsequent push operation attempts to increment `tos` beyond its range. The lack of formal semantics implies that the tasks are free to choose their own semantics for this increment. Unfortunately, different tasks might assign different semantics. This is clearly undesirable since the tasks do not have a consistent interpretation of the specification. However, assume for the moment that an s sized register defines a modulo s arithmetic (this is the semantics imposed by the Cadence Verilog-XL simulator.). In that case `tos` will point to location 0 for a full stack. Such a wraparound stack cannot distinguish between a full and empty stack.

```
module StationaryDataSpec(dataOut, dataIn, op, clock)
    output dataOut;
    input [0:1] op;    // Push=2'b00, Pop=2'b01, Hold=2'b10, Illegal=2'b11
    input dataIn;
    input clock;

    reg [0:1023] Sstack;    // Defining a stack of 1024 elements.
    reg dataOut;
    reg [0:9] tos;    // 10 bit top of stack pointer.

    initial tos = 0;

    always @(posedge clock)
    begin
        if (op == 2'b00)    // Push Operation.
        begin
            Sstack[tos] = dataIn;
            tos = tos + 1;
        end

        if (op == 2'b01)    // Pop Operation.
        begin
            tos = tos - 1;
            dataOut = Sstack[tos];
        end
    end
endmodule
```

Figure 2.2: A Verilog HDL specification of a stationary data stack.

Note that the two stack specifications define different boundary conditions. To illustrate the difference assume that both stacks are full with a logic 0 stored at the bottom of the each stack and logic 1 stored at all other locations. Now empty both stacks by performing 1024 pop operations. The input/output behavior for these 1024 operations will be the same for both specifications. At the end

of the pop operations, however, the internal state of the two stacks are different. The internal state of the stationary stack is exactly the same as before. The moving data stack, however, now has a logic 0 stored in each location of the stack. Two subsequent pop operations from the empty stack will reflect this difference in the input/output behavior of the stacks. The problem again is that the stacks have been overspecified. The details of the boundary conditions might differ from one realization to the other. Unless the designer requires a specific way of handling these boundary conditions, they should not be specified.

Figure 2.3: Partial order defined on logic 0,1,X.

It should be acknowledged that overspecification is not an inherent limitation of these HDLs. It is possible to strictly specify a stack through a judicious use of the X logic value. In this model, the logic value X is assumed to imply less information than the logic 0 and logic 1 values as shown in Figure 2.3. The operations used in the specification have to be monotonic in the following sense: If an operation evaluates to a 0 or 1 in the presence of some X's, then the operation should evaluate to the same value even if any number of X's are replaced by any combination of 0's and 1's. Under this monotonicity constraint, the stationary stack can be strictly specified as shown in Figure 2.4. The registers have been set to X for the boundary conditions and the illegal operation. The tasks are free to interpret X as either 0 or 1. The irony in such a description is that the designer had to give careful consideration to those aspects that contribute the least to the functionality of the stack, i.e. the boundary conditions and the illegal operation. It can be seen that this description is much more complex and cumbersome compared to the simple stationary stack specification in Figure 2.2. Loops had to be introduced in the description. Only upon a careful analysis does one realize that this description specifies a stationary stack.

```
module StrictlySpecifiedStationaryDataStackSpec(dataOut, dataIn, op, clock);
    output dataOut;
    input [0:1] op;   // Push=2'b00, Pop=2'b01, Hold=2'b10, Illegal=2'b11
    input dataIn;
    input clock;

    reg [0:1023] Sstack;   // Defining a stationary stack of 1024 elements.
    reg dataOut;
    reg [0:10] tos;   // 11 bit top of stack pointer.
    integer i;

    initial tos = 0;

    always @(posedge clock)
    begin
        if (op == 2'b00)   // Push Operation.
        begin
            if (tos < 1024)
            begin
                Sstack[tos] = dataIn; tos = tos + 1;
            end
            else begin   // Push on a full stack is not defined.
                for (i = 0; i < 1024; i = i + 1) Sstack[i] = 1'bx;
                tos = 11'bx;
            end
            dataOut = 1'bx;
        end

        if (op == 2'b01)   // Pop operation.
        begin
            if (tos > 0)
            begin
                tos = tos - 1; dataOut = Sstack[tos];
            end
            else begin   // Pop from an empty stack is not defined.
                for (i = 0; i < 1024; i = i + 1) Sstack[i] = 1'bx;
                tos = 11'bx;
                dataOut = 1'bx;
            end
        end

        // Hold operation implicitly defined by the semantics.

        if (op == 2'b11)   // Illegal operation.
        begin
            for (i = 0; i < 1024; i = i + 1) Sstack[i] = 1'bx;
            tos = 11'bx;
            dataOut = 1'bx;
        end
    end
endmodule
```

Figure 2.4: A strictly specified Verilog HDL description of a stationary data stack.

In summary, Hardware Description Languages are not ideal as a specification notation for formal verification due to the following reasons:

- **Realization Dependent**: Probably the most important limitation is that low-level implementation details creep into the high-level behavioral HDL specification. This makes it extremely difficult to write a specification which is independent of the realization. The need for a specification language to be independent of the realization has been referred to as the *Neutrality* property[76].
- **Simulation Specific**: HDLs are very simulation specific. All these languages were originally developed to allow users to construct simulation models. Hence, they contain features that are based strongly on the execution model provided by event-driven simulators.
- **Sequencing**: A designer has to give careful consideration to sequencing of operations while writing HDL descriptions. This might be overkill since the verification task is usually concerned only with the end result of a set of operations.
- **OverSpecification**: Designers normally tend to overspecify their systems. The lack of don't care information results in false negative reports by the formal verification task. The false negative reports are due to the verifier labeling a correct design as defective. A stack might be reported to be defective due to different boundary conditions or a different response to an illegal operation.
- **Formal Semantics**: Most HDLs do not have well-defined formal semantics. Even though UDL/I has documented formal semantics, the semantics are defined in terms of an event-driven scheme, which takes us back to the point that HDLs are very simulation specific.

It seems that Hardware Description Languages are appropriately named. They are languages for describing hardware. However, they are not appropriate for specifying hardware. In this thesis, we introduce a *Hardware Specification Language* to specify hardware.

2.2 Related Work

Our Hardware Specification Language is based on the State Machine Assertion Language (SMAL) introduced by Beatty[13]. SMAL is a declarative language with a documented denotational

semantics. The language describes each operation as an assertion. The assertion consists of a precondition describing the set of abstract states at the start of the operation and a postcondition describing the set of abstract states at the end of the operation. SMAL was used to describe a partial specification of the Hector microprocessor[59].

2.3 Basic Form of Hardware Specification Language

This section defines the basic form of the Hardware Specification Language. The exact syntax of the language will be described in later sections. The basic form is quite simple. It might seem that this basic form can only be used to specify trivial behaviors. Later on, however, we will extend the language into the symbolic and multivalued domains and give several examples of how nontrivial behavior can be specified using this language.

The abstract specification is associated with a set of single-bit *abstract elements*, S_v. Abstract elements collectively refer to inputs, outputs, and state elements in the abstract model. The effect of each operation is defined as an *abstract assertion*. Each abstract assertion is of the form: $P \Rightarrow Q$, which should be read as the precondition P leads to the postcondition Q. There is an implicit notion of passage of time from the precondition to the postcondition. P and Q are *abstract formulas* of the form:

- **Simple abstract formula:** $(s_i \text{ is } 0)$ and $(s_i \text{ is } 1)$ are abstract formulas if $s_i \in S_v$.
- **Conjunction:** $(F_1 \text{ and } F_2)$ is an abstract formula if F_1 and F_2 are abstract formulas.
- **Domain Restriction:** $(0 \rightarrow F)$ and $(1 \rightarrow F)$ are abstract formulas if F is an abstract formula.

In addition, we will let *true* represent the trivially true abstract formula. The domain restriction appears at first somewhat strange. The usefulness will become apparent when we extend the abstract formulas to a symbolic domain.

Each abstract assertion defines a set of transitions in an abstract Moore machine. The semantics of an abstract assertion can be defined as follows: If the abstract machine starts in a set of states that satisfy the precondition, then the machine should transition into a set of states that satisfy the postcondition. Else, if the machine starts in a set of states that does not satisfy the precondition, then

the assertion does not place any restrictions on the set of transitions. The intersection of all assertions defines a nondeterministic Moore machine. In this sense, the abstract specifications are theoretically being represented as Moore machines. The assertions are simply a convenient way to describe the set of transitions of this machine.

Consider the example of an inverter. An inverter requires two abstract elements, `In` and `Out`, where `In` corresponds to the input of the inverter and `Out` corresponds to the output of the inverter. The inverter can be specified using 2 abstract assertions:

(`In` is 0) ➡ (`Out` is 1) // Assertion A_1

(`In` is 1) ➡ (`Out` is 0) // Assertion A_2

Mathematically, an abstract assertion can be defined as follows: Assume S is the set of assignments to the abstract elements, $S = \{0,1\}^n$, where $n = |S_v|$. Define a satisfying set for an abstract formula as the set of assignments that satisfy the abstract formula. The satisfying set of an abstract formula F, written $Sat(F)$, is defined recursively:

- $Sat(true) = S$.
- $Sat(s_i \text{ is } 0)$ is the subset of S containing 2^{n-1} assignments having the i^{th} position as 0. Similarly $Sat(s_i \text{ is } 1)$ is the subset of S containing 2^{n-1} assignments having the i^{th} position as 1.
- $Sat(F_1 \text{ and } F_2) = Sat(F_1) \cap Sat(F_2)$.
- $Sat(0 \rightarrow F) = S$, and $Sat(1 \rightarrow F) = Sat(F)$

For an abstract assertion $A = P \Rightarrow Q$, the sets $Sat(P)$ and $Sat(Q)$ are the set of assignments that satisfy the precondition and postcondition respectively. The transitions associated with the assertion can now be defined as: $Trans(A) = [Sat(P) \times Sat(Q)] \cup [(S - Sat(P)) \times S]$. Remember if the abstract machine starts in a set of states that satisfies the precondition, then the machine should transition into a set of states that satisfies the postcondition. The set $Sat(P) \times Sat(Q)$ represents the set of transitions when the precondition is satisfied. If the machine starts in a set of states that does not satisfy the precondition, then the assertion does not

place any restrictions on the set of transitions. The set $(S - Sat(P)) \times S$ represents the set of transitions when the precondition is not satisfied.

Let i index the set of abstract assertions. The intersection $\bigcap_i Trans(A_i)$ defines a nondeterministic Moore machine corresponding to the abstract specification.

For the inverter example, $n = 2$ and the set S has 4 state assignments, $S = \{00, 01, 10, 11\}$, where the left bit is the logic value associated with `In` and the right bit is the logic value associated with `Out`. The transitions corresponding to these assertions are shown in Figure 2.5. Let us consider assertion A_1. Now $Sat(\texttt{In is 0}) = \{00, 01\}$ and $Sat(\texttt{Out is 1}) = \{01, 11\}$. The solid edges represent transitions when the precondition is satisfied, i.e., the set $Sat(P) \times Sat(Q)$. The shaded edges represent the transitions when the precondition is not satisfied, i.e., the set $(S - Sat(P)) \times S$. A similar case can be made for assertion A_2. The intersection of the transitions associated with both assertions gives the Moore model for the specification. In this particular model, nondeterminism is used only to represent the choice of the next input. In other cases, nondeterminism can be used to represent unspecified or don't care behavior such as output of the multiplier while performing an add operation in a processor.

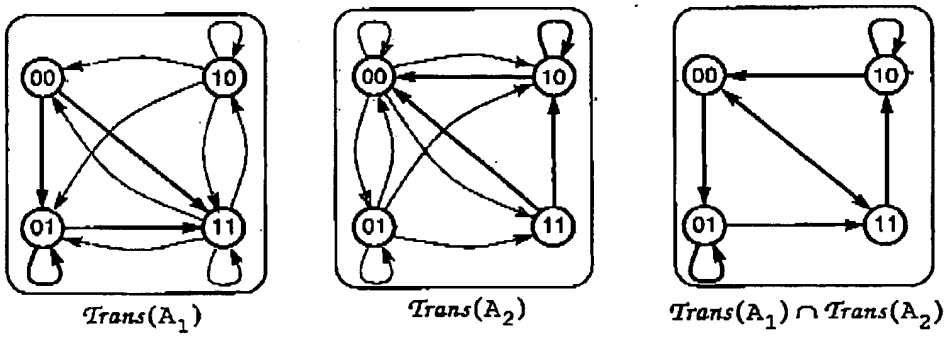


Figure 2.5: Nondeterministic Moore machine model for an inverter specification.

The transition relation corresponding to an abstract assertion can be expressed in terms of a characteristic function. For every state variable $s_i \in S_v$, introduce 2 variables denoting the present and next-state values for the state variable. Let the Boolean variables p_i and n_i denote the present and next-state values respectively. Define a Boolean function, $\mathcal{R}el(F)$, for an abstract formula F. The function $\mathcal{R}el(F)$ can be defined recursively[1]:

- $\mathcal{R}el(true) = 1$
- For simple abstract formulas appearing in the precondition, $\mathcal{R}el(s_i \text{ is } 0) = \overline{p_i}$ and $\mathcal{R}el(s_i \text{ is } 1) = p_i$. For simple abstract formulas appearing in the postcondition, $\mathcal{R}el(s_i \text{ is } 0) = \overline{n_i}$ and $\mathcal{R}el(s_i \text{ is } 1) = n_i$.
- $\mathcal{R}el(F_1 \text{ and } F_2) = \mathcal{R}el(F_1) \cdot \mathcal{R}el(F_2)$.
- $\mathcal{R}el(0 \rightarrow F) = 1$, and $\mathcal{R}el(1 \rightarrow F) = \mathcal{R}el(F)$.

For an abstract assertion, $A = P \Rightarrow Q$, the characteristic function for the transition relation corresponding to A is defined as $\chi(A) = \overline{\mathcal{R}el(P)} + \mathcal{R}el(Q)$.

Let i index the set of abstract assertions. The conjunction of all $\chi(A_i)$ is the characteristic function for the transition relation corresponding to the abstract specification.

Consider the example of the inverter. Let p_{In} and n_{In} be the present and next-state Boolean variables corresponding to the abstract element In. Similarly, let p_{Out} and n_{Out} be the Boolean variables corresponding to Out. The characteristic functions corresponding to assertions A_1 and A_2 are $\chi(A_1) = p_{In} + n_{Out}$ and $\chi(A_2) = \overline{p_{In}} + \overline{n_{Out}}$. Therefore, the characteristic function corresponding to the specification is $\chi(A_1) \cdot \chi(A_2) = p_{In} \oplus n_{Out}$.

2.4 Extensions to Hardware Specification Language

2.4.1 Symbolic Extension

The preceding section defined the scalar version of an abstract assertion. A *symbolic abstract assertion* can be used to define a set of scalar abstract assertions. Each abstract assertion can be associated with a set of symbolic variables. All Boolean operations ($^-, \cdot, +, \oplus, \bar{\oplus}$) are extended from the scalar to the symbolic domain.

A symbolic abstract assertion is associated with a set of symbolic variables V. A symbolic abstract assertion is of the form: $A = P \Rightarrow Q$, where P and Q are symbolic abstract formulas of the form:

1. $^-, \cdot, +, \oplus, \bar{\oplus}$ represent the Boolean negation, conjunction, disjunction, exclusive OR, and exclusive NOR operators respectively.

- **Simple abstract formula:** $(s_i \text{ is } 0)$ and $(s_i \text{ is } 1)$ are symbolic abstract formulas if $s_i \in S_v$.
- **Conjunction:** $(F_1 \textbf{ and } F_2)$ is a symbolic abstract formula if F_1 and F_2 are symbolic abstract formulas.
- **Domain Restriction:** $(e \rightarrow F)$ is a symbolic abstract formula if F is a symbolic abstract formula and e is a Boolean expression over V.

Note that the only change from the definition of a scalar abstract assertion is that the domain constraint can now be a Boolean expression rather than only 0 or 1.

We introduce the notation $(s_i \text{ is } e)$ as a shorthand for the formula $(\bar{e} \rightarrow (s_i \text{ is } 0)) \textbf{ and } (e \rightarrow (s_i \text{ is } 1))$. That is, we constrain state element s_i to have the particular symbolic Boolean value denoted by the expression e.

As an example, a single symbolic assertion can specify an inverter as shown below. The variable a is a symbolic variable and "¯" is the negation operator extended to the symbolic domain. The symbolic assertion captures both assertions A_1 and A_2 corresponding to values 0 and 1 for the variable a.

(In **is** a) ➡ (Out **is** $\bar{a}$)

As another example, consider an OR operation. Let InA and InB be the inputs and Out be the output for the OR operation. The OR operation can be specified with a single symbolic abstract assertion as shown below. The variables a and b are symbolic variables and "|" is the disjunction operator extended into the symbolic domain. The symbolic abstract assertion captures 4 scalar assertions corresponding to 4 (i.e. 00,01,10,11) possible values for variables a and b.

(InA **is** a) **and** (InB **is** b) ➡ (Out **is** a|b)

A symbolic abstract assertion can be converted into the corresponding transition relation. As before, for every state variable $s_i \in S_v$, introduce 2 variables denoting the present and next-state values for the state variable. Let the Boolean variables p_i and n_i denote the present and next state values respectively. Extend the Boolean function, $\mathcal{R}el(F)$, for a symbolic abstract formula as follows:

- $\mathcal{R}el(true) = 1$.
- For simple abstract formulas appearing in the precondition, $\mathcal{R}el(s_i \text{ is } e) = p_i \oplus e$. For simple abstract formulas appearing in the postcondition, $\mathcal{R}el(s_i \text{ is } e) = n_i \oplus e$.
- $\mathcal{R}el(F_1 \text{ and } F_2) = \mathcal{R}el(F_1) \cdot \mathcal{R}el(F_2)$.
- $\mathcal{R}el(e \rightarrow F) = \bar{e} + \mathcal{R}el(F)$.

For a symbolic abstract assertion, $A = P \Rightarrow Q$, the characteristic function for the transition relation corresponding to A is defined as $\chi(A) = \forall_V (\overline{\mathcal{R}el(P)} + \mathcal{R}el(Q))$, where $\forall$ is the universal quantification operator and V is the set of symbolic variables.

As an example, consider the OR operation. Let p_{InA} and n_{InA} be the Boolean variables introduced for state element InA. Also, let p_{InB} and n_{InB} be the Boolean variables introduced for state element InB. And p_{Out} and n_{Out} be the Boolean variables introduced for state element Out. The characteristic function for the transition relation corresponding to the assertion is:

$$\forall_{a,b} \overline{\left(p_{InA} \oplus a\right) \cdot \left(p_{InB} \oplus b\right)} + n_{Out} \oplus (a+b) = (p_{InA} + p_{InB}) \oplus n_{Out}$$

2.4.2 Vector and Data Handling Extensions

So far, the abstract elements have been limited to single bit and the symbolic variables have been limited to Boolean variables. The Hardware Specification Language has several extensions to ease the task of writing abstract specifications. The abstract elements are extended to represent different word sizes. A type definition section can be used to define a bit-vector, enumerations, and multidimensional array types. Types are used to define abstract elements and symbolic variables. *Symbolic Indexing* can be used to index into array types. Expressions over the symbolic variables can be defined using logical, bitwise, arithmetic, and relational operators.

Consider the example of a bitwise-OR operation for a 32-bit word size. Define three 32-bit abstract elements, SA, SB, and ST. State elements SA and SB serve as the source operands and ST serves as the target operand. Define two 32-bit symbolic variables a and b that denote the current values of the SA and SB respectively. The bitwise-OR operation can be captured with a single symbolic abstract assertion:

(SA is a) and (SB is b) ➡ (ST is a|b)

The abstract formula (SA is a) is a short form for the abstract formula (SA[0] is a0) and (SA[1] is a1) and ... and (SA[31] is a31) . The symbolic abstract assertion captures 4^{32} scalar assertions corresponding to all possible combination of assignments to variables a and b.

2.5 Syntax and Examples

The syntax of the Hardware Specification Language is described in Appendix A. This section discusses some examples.

2.5.1 Bitwise-OR Operation

The specification for a bitwise-OR operation on 32-bit source and target operands is shown in Figure 2.6. The specification can be divided into 3 separate sections: **1.** Type Definitions, **2.** State Declarations **3.** Set of Assertions. The type definition section defines a set of types required for the specification. The specification for the bitwise-OR operation defines a word as an array of 32 bits. The state declaration section defines the source and target operands. The assertion defines a bitwise-OR operation. The assertion is associated with symbolic variables. The keyword LEADSTO separates the postcondition and precondition and corresponds to the symbol "➡" introduced earlier.

```
MACHINE BitwiseOr;

   TYPE WORD is ARRAY(31 to 0) of BIT;
   STATE SA, SB, ST: WORD;

   Assertion {
      VARIABLE a, b: WORD;

      (SA is a) and (SB is b)
        LEADSTO
      (ST is a | b)
   }

ENDMACHINE
```

Figure 2.6: Abstract specification for a bitwise-OR operation.

2.5.2 Stack

The complete specification for a stack is shown in Figure 2.7. The type definition section defines an enumerated type for the push, pop, and hold operations on the stack. The state declaration section defines the abstract elements associated with the stack. The stack is defined as an array of 1024 elements. The abstract element `depth` maintains a count of the number of items currently in the stack. The set of assertions defines the push, pop, and hold operations on the stack.

Consider the assertion for the push operation. The `WHEN` keyword is used to define domain restrictions. Note that a global domain restriction is placed on the push assertion. The global domain restriction ensures that the assertion does not specify the behavior when someone tries to push into a full stack. In this sense, the push assertion is ensuring that the system is not overspecified. The realization is free to pick what it wants to do for a push on a full stack. The push assertion should be read as follows: If in the precondition, `op` describes a push operation, `dataIn` has some value (say `v`), stack has currently `d` elements, and location `i` of the stack has some value `u`, then in the postcondition, the top of stack should get value `v`, location `i+1` of the stack should get value `u`, and the stack finally has `d+1` elements.

A similar case can be made for push and hold assertions. Note that the Verilog description in Figure 2.4 specified the illegal operation but not the hold operation. The HSL, on the other hand, specifies the hold operation but not the illegal operation. The reason is due to different semantics for the two descriptions. The Verilog semantics is that if something is not specified or updated then it retains the previous state. The HSL semantics, however, is that if something is not specified, then anything is fair game. Therefore, the illegal operation is not specified. The hold operation, however, is explicitly specified to capture the fact that the stack should retain previous state.

The specification succinctly captures our abstract notion of a stack.

```
MACHINE stack;

   TYPE operations is (push, pop, hold);

   STATE dataIn, dataOut: BIT;
   STATE control: operations;
   STATE depth: RANGE 0 to 1024;
   STATE stack: ARRAY (1023 to 0) of BIT;

   PushAssertion {
      VARIABLE i: RANGE 0 to 1023;
      VARIABLE d: RANGE 0 to 1024;
      VARIABLE u, v: BIT;

      WHEN (d < 1024)
        (control is push) and (dataIn is v) and (depth is d) and (when (i < d) stack[i] is u)
          LEADSTO
        (stack[0] is v) and (depth is d+1) and (when (i < d) stack[i+1] is u)
   }

   PopAssertion {
      VARIABLE i: RANGE 0 to 1023;
      VARIABLE d: RANGE 0 to 1024;
      VARIABLE u, v: BIT;

      WHEN (d > 0)
        (control is pop) and (depth is d) and (stack[0] is v) and (when (0 < i < d) stack[i] is u)
          LEADSTO
        (dataOut is v) and (depth is d-1) and (when (0 < i < d) stack[i-1] is u)
   }

   HoldAssertion {
      VARIABLE i: RANGE 0 to 1023;
      VARIABLE d: RANGE 0 to 1024;
      VARIABLE u: BIT;

      (control is hold) and (depth is d) and (stack[i] is u)
        LEADSTO
      (depth is d) and (stack[i] is u)
   }

ENDMACHINE
```

Figure 2.7: Abstract specification for a stack.

2.5.3 Addressable Accumulator

Consider the example of an addressable accumulator shown in Figure 2.8. The accumulator can maintain the sum of signals for 16 different channels, storing the sums in the register array. There are two possible operations. The store operation stores the value on the data input in the addressed register. The add operation takes the sum of the data input and the addressed register and stores it back in the addressed register.

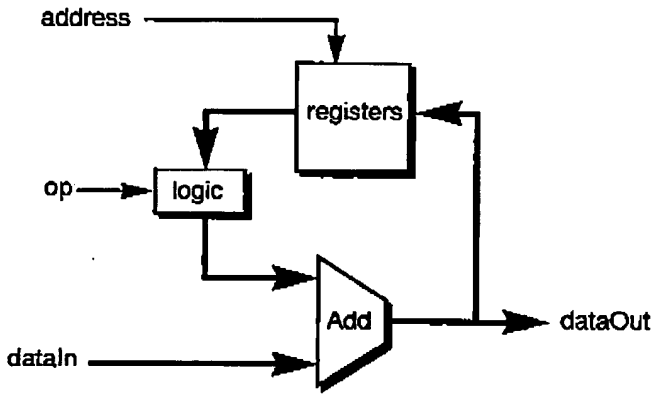


Figure 2.8: Addressable accumulator

The abstract specification for the addressable accumulator is shown in Figure 2.9. The type definition and state declaration sections define the data path to be 32 bits wide. A 4-bit address is required to address 16 channels in the register array. Assertions define the store and add operations. An additional assertion is required to indicate that the rest of the registers (other than the addressed register) should maintain their state.

```
MACHINE accumulator;

    TYPE DATAWORD is ARRAY(31 to 0) of BIT;
    TYPE ADDRWORD is ARRAY(3 to 0) of BIT;
    TYPE OPS is (store, add);

    STATE op: OPS;
    STATE address: ADDRWORD;
    STATE dataIn, dataOut: DATAWORD;
    STATE registers: ARRAY (15 to 0) of DATAWORD;

    StoreAssertion {
        VARIABLE i: ADDRWORD;
        VARIABLE a: DATAWORD

        (op is store) and (dataIn is a) and (address is i)
          LEADSTO
        (dataOut is a) and (registers[i] is a)
    }

    AddAssertion {
        VARIABLE i: ADDRWORD;
        VARIABLE a, b: DATAWORD;

        (op is add) and (dataIn is a) and (address is i) and (registers[i] is b)
          LEADSTO
        (dataOut is a+b) and (registers[i] is a+b)
    }

    MaintainStateAssertion {
        VARIABLE i, j: ADDRWORD;
        VARIABLE b: DATAWORD;

        WHEN (i != j)
          (address is i) and (registers[j] is b)
            LEADSTO
          (registers[j] is b)
    }

ENDMACHINE
```

Figure 2.9: Abstract specification of an addressable accumulator.

2.6 Miscellaneous Issues

The abstract specification defines a nondeterministic Moore machine. This opens the possibility of performing a high-level verification of the abstract specification as is done in the *Symbolic Model Verifier* (SMV)[33]. The abstract specification would serve as the state machine. Properties would be expressed in *Computation Tree Logic* (CTL)[30]. Symbolic model checking could then be used to verify properties of our abstract specification. Therefore, there are two different levels of verification. A high-level verification verifies properties of our abstract specification. And then a low-level verification is used to verify that the circuit fulfills the abstract specification.

THIS PAGE BLANK (USPTO)

Chapter 3

Implementation Mapping

The abstract specification describes the high-level behavior of the system independent of any timing or implementation details. Implementation specific details are captured in the mapping. As an example, the abstract specification describes the instruction set architecture of a processor. The details of the underlying implementation such as the pipeline structure, forwarding logic, pipeline interlocks, multiple instruction issue, and interface protocols are captured in the mapping.

Our implementation mapping is defined as a set of *map machines*. A map machine is used to describe the temporal and spatial mapping for each abstract element in the specification. The mapping relates the abstract element to a set of signals and internal nets in the circuit. Abstract inputs and outputs can be mapped into protocols on a set of interface signals in the circuit. Abstract state elements are mapped into a set of circuit state elements. A single map machine is used to map all instances of the abstract state element in the assertion. For an abstract state element in the precondition, the map machine defines a set of constraints on the internal state elements and inputs in the circuit. For the abstract state element in the postcondition, the same map machine defines a set of allowed responses and state transitions.

3.1 Related Work

Beatty laid down the foundation for the implementation mapping[13]. Each abstract element was mapped into a set of signals in the circuit as shown in Figure 3.1. An abstract input was mapped into a sequence of logic values on a set of circuit inputs. An abstract output was mapped into a sequence of logic values on a set of circuit outputs. An abstract state was mapped into a sequence of logic values on a set of circuit states. The abstract state can appear both in the precondition and postcondition of an assertion. The user defines a single mapping for the abstract state. For the abstract state in the precondition, the mapping defines a set of constraints on the circuit states. For

the abstract state in the postcondition, the mapping defines the set of acceptable logic values on the circuit states.

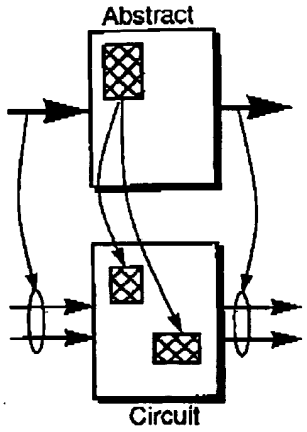


Figure 3.1: Mapping the abstract model to the circuit model.

Beatty, however, could express only a very limited set of temporal behaviors. The mapping language was formulated in terms *marked strings*, which provided a formal model for aligning and overlapping timing diagrams. Marked strings were defined over an alphabet. The alphabet denoted logic values for signals in the timing diagram. The marks denoted the start and end of an operation and could be used to align and overlap successive operations. Marked strings could express only bounded single behavior sequences. Moreover, Beatty mapped abstract inputs and outputs into a set of unidirectional signals in the circuit. Several systems require abstract inputs and outputs to be mapped into protocols on a set of bidirectional signals as shown in Figure 3.2.

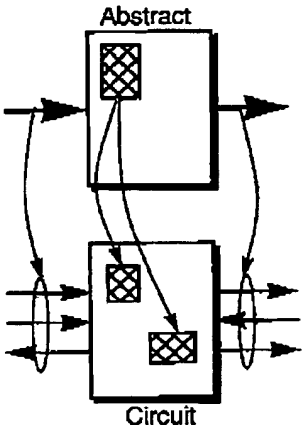


Figure 3.2: Extending the mapping for bidirectional inputs and outputs.

Our implementation mapping is defined in terms of a variation of state diagrams called *control graphs*. State diagrams allow users to define unbounded and divergent behavior. The user can create an environment around the circuit and define complex nondeterministic interface protocols.

3.2 Control Graphs

Control graphs are state diagrams with the capability of synchronization at specific time points. There are two types of vertices in a control graph: **1.** *Event vertices* representing instantaneous time points. **2.** *State vertices* representing some non-zero duration of time. Event vertices are used for synchronization. A control graph has a unique event vertex with no incoming edges called the source vertex and a unique event vertex with no outgoing edges called the sink vertex. Nondeterminism is modeled as multiple outgoing edges from a vertex. In general, a control graph can be viewed as a "sausage" as shown in Figure 3.3. The source and sink vertices serve as two ends of the sausage. There exists a path from the source to each state vertex in the control graph and from each state vertex to the sink of the control graph.

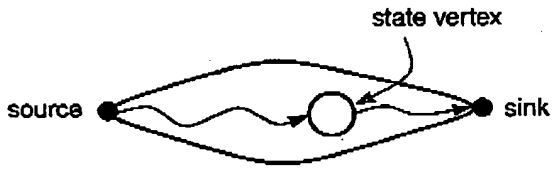


Figure 3.3: General form of a control graph.

As an example, consider the control graph shown in Figure 3.4. Vertices `s`, `u` and `t` are event vertices with `s` as the source vertex and `t` as the sink vertex. Vertices v_1 and v_2 are state vertices. The control graph represents that the system can be in vertex v_1 for an arbitrary number of times (0 or more) before transitioning to vertex v_2.

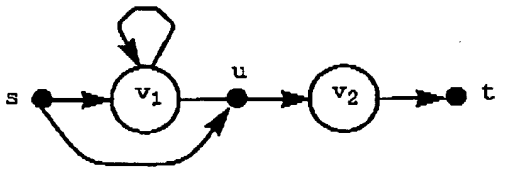


Figure 3.4: Example control graph.

Mathematically, a control graph can be represented as a tuple: $G = \langle V, U, E, s, t \rangle$, where

- V is the set of *state vertices*.
- U is the set of *event vertices*.
- E is the set of directed edges.
- s is the *source*, $s \in U$.
- t is the *sink*, $t \in U$.

3.3 Basic Form of Implementation Mapping

This section defines the basic form of the implementation mapping. The exact syntax of the mapping will be described in later sections. The basic form of the mapping corresponds to the basic form of the abstract specification. Later on, we will extend the mapping language into the symbolic and vector domains and give several examples.

The mapping uses a *main machine* to define the flow of control for individual system operations. The main machine describes the pipeline structure for the implementation. *Map machines* are used to define the mapping for each abstract element. The map machines are synchronized with the main machine at specific time points to relate the flow of control of individual abstract elements with the flow of control of the entire operation.

The implementation mapping has four components:

- Set of *node elements*.
- Hierarchy of state vertices.
- Single *main machine*.
- Set of *map machines*.

The set of node elements N_v represents a subset of nets in the realization. The node elements represent inputs, outputs, and internal nets in the circuit. The node elements are used to define *node formulas*. Node formulas are used in defining the hierarchy of state vertices and the set of map machines and are of the form:

- **Simple node formula:** $(n_i \text{ is } 0)$ and $(n_i \text{ is } 1)$ are node formulas if $n_i \in N_v$.
- **Conjunction:** $(F_1 \text{ and } F_2)$ is a node formula if F_1 and F_2 are node formulas.
- **Domain Restriction:** $(0 \rightarrow F)$ and $(1 \rightarrow F)$ are node formulas if F is a node formula.

In addition, we shall refer to *true* as a trivially true node formula. The domain restriction appears at first somewhat strange. The usefulness will become apparent when we extend node formulas to the symbolic domain.

The next few sections describe each component of the implementation mapping in detail.

3.3.1 Hierarchy of State Vertices

State vertices can be defined at varying levels of abstraction. The basic state vertex is a *phase-level vertex*. A phase-level vertex represents a period of time in which all external inputs are held fixed and the system operates until it reaches a stable state. Over and above the phase-level vertex, the user can define a hierarchy of state vertices. The hierarchy corresponds to the temporal hierarchy in the circuit. Each level in the hierarchy defines a state vertex as a control graph in terms of lower level state vertices. State vertices can be associated with node formulas, which are typically used to define the clocking scheme for the circuit.

As an example, consider a realization that uses two clocks, φ_1 and φ_2, and a 4-phase non-overlapping clocking scheme. A clock cycle, therefore, corresponds to 4 clock phases. A cycle-level state vertex can be defined as a control graph shown in Figure 3.5. Each state vertex in the control graph is a phase-level vertex. The node formulas, shown in shadowed boxes, define the values of the clock signals in each clock phase.

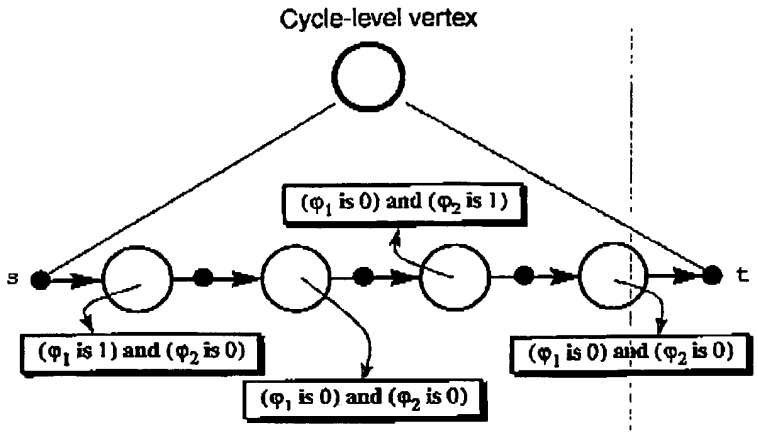


Figure 3.5: Defining a cycle-level state vertex.

Above a cycle-level vertex, the user can define an instruction-level vertex. Consider a processor where each instruction takes 2 cycles. An instruction-level vertex can be defined as shown in Figure 3.6. Each instruction-level vertex corresponds to 2 cycle-level vertices or 8 phase-level vertices.

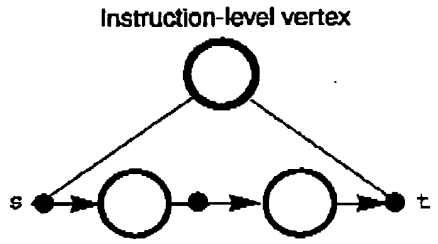


Figure 3.6: Defining an instruction-level vertex.

Mathematically, a hierarchical state vertex is a tuple of the form: $\langle V_h, U_h, E_h, s_h, t_h, \sigma_h \rangle$, where

- V_h is the set of lower-level state vertices. At the lowest level, V_h is the set of phase-level state vertices.
- $\langle V_h, U_h, E_h, s_h, t_h \rangle$ is the control graph.
- σ_h is the labelling function that labels state vertices with node formulas.

3.3.2 Main Machine

The main machine defines the flow of control for individual system operations. The pipeline structure of the implementation is captured in the main machine. The main machine can be viewed as a control graph with a *nextmarker function*, as shown in Figure 3.7. The source of the control graph denotes the start of the operation and the sink denotes the end of the operation. The control graph can be viewed as series of subgraphs. Each subgraph represents a pipeline stage. The nextmarker function relates an event vertex in a pipeline stage with a corresponding event vertex in the next pipeline stage. Typically, the nextmarker function defines the start and end of each pipeline stage.

Figure 3.7: General form of a main machine.

As an example, consider a simple processor with fetch, decode, and execute pipeline stages. Assume that all instructions spend a single clock cycle in each pipeline stage. The main machine for such a processor is shown in Figure 3.8. The vertices F, D, and E are cycle-level state vertices representing the fetch, decode, and execute stages respectively.

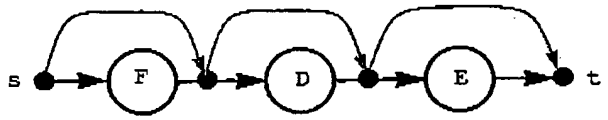


Figure 3.8: Example 1. Main machine for a simple processor.

The nextmarker defines how individual operations can be stitched together to create execution sequences. Individual instructions can be stitched together to create execution sequences as shown in Figure 3.9. The machines M^1, M^2, M^3 are multiple copies of the main machine with the superscripts denoting successive instructions. The machine M^* corresponds to the infinite execution sequence obtained from stitching all instructions. Stitching is performed by aligning an event vertex of an instance of the main machine with the corresponding nextmarker event in the previous main machine and *composing* the machines. In this case, composition is simply the cross-product

of all the machines. Therefore, the state vertex $E^1D^2F^3$ in machine M^* corresponds to the cross product of state vertices E^1 in machine M^1, D^2 in machine M^2, and F^3 in machine M^3.

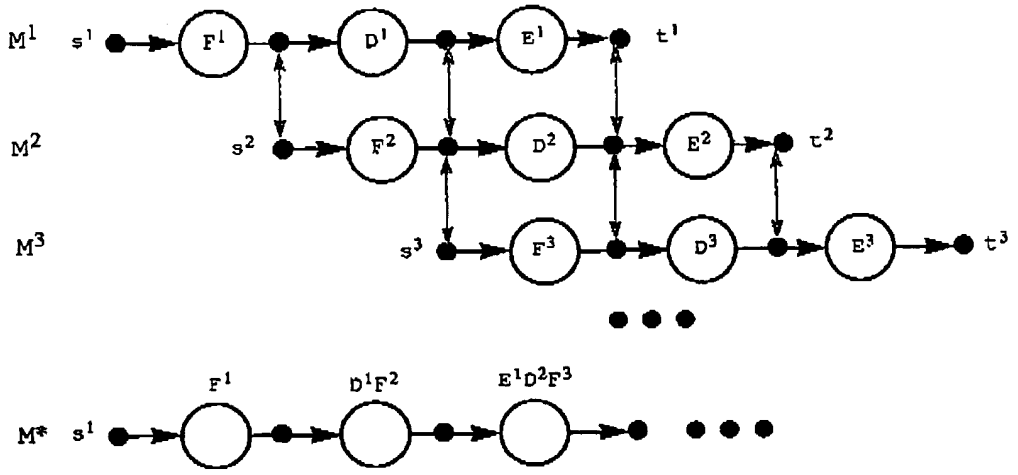


Figure 3.9: Example 1. Stitching main machines together to form execution sequences.

Mathematically, the main machine is a tuple of the form: $M = \langle V_m, U_m, E_m, s_m, t_m, \beta_m \rangle$, where

- $\langle V_m, U_m, E_m, s_m, t_m \rangle$ is the control graph.
- β_m is the nextmarker function, $\beta_m : U_m \rightarrow U_m$. This is a partial function.

The nextmarker function, β_m, is a partial function. Only a few event vertices will be in the inverse image of a nextmarker function. The rest of the event vertices will be considered as *free* vertices. For a free vertex, $u_m \in U_m$, the nextmarker function is defined to be $\beta_m(u_m) = \Lambda$. Strictly speaking, the nextmarker function should be of the form: $\beta_m : U_m \rightarrow U_m \cup \Lambda$. However, for the sake of simplicity, we will use $\beta_m : U_m \rightarrow U_m$, with the understanding that Λ represents a free vertex.

The nextmarker function places some restrictions on the control graph. The event vertex $\beta_m(u_m)$ should be a vertex cut of the control graph. The vertex cut divides the set of vertices into 2 sets, i.e., Y_m and Z_m. The set Y_m represents the set of vertices to the left of the vertex cut and the set Z_m represents the set of vertices to the right of the vertex cut. There cannot be any edges from vertices in Z_m to either $\beta_m(u_m)$ or vertices in X_m. The restriction ensures that the pipeline is committed to an instruction once it has started.

A nextmarker has to be defined for the source vertex s_m. The event vertex $\beta_m(s_m)$ is the time point where the next instruction can be started. For a nonpipelined circuit, $\beta_m(s_m) = t_m$, since the current operation must end before the next operation can start. For pipeline circuits, however, the event vertex $\beta_m(s_m)$ is a cutset between source and sink. The cutset defines the time point where the next operation can be started, thus overlapping the current and next operations.

Let us consider another example where a processor has two pipeline stages, namely, the fetch and execute stage. Further assume that an instruction might spend an arbitrary number of cycles (1 or more) in each stage. The main machine for such a processor is shown in Figure 3.10.

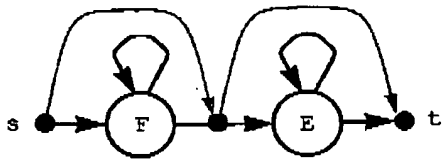


Figure 3.10: Example 2. Main machine for a processor.

The nextmarker function defines how instructions can be stitched to create execution sequences. Let us consider two instructions, M^1 and M^2. They get aligned as shown in Figure 3.11. Unfortunately, this is an overly restrictive alignment, since it requires the execute E^1 and fetch F^2 to finish simultaneously.

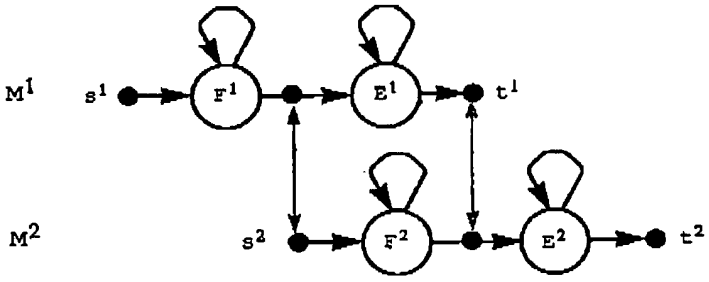


Figure 3.11: Example 2. Alignment of main machines.

The restriction can be removed by describing a slightly more complicated main machine. The main machine has to be augmented with a dummy vertex as shown in Figure 3.12. The dummy vertex represents the situation where the execute stage of the current instruction finished before the fetch stage of the next instruction.

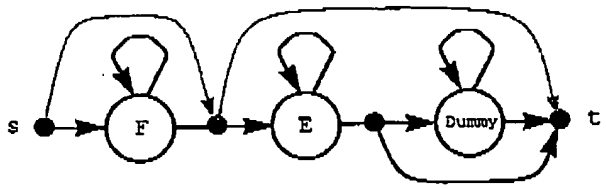


Figure 3.12: Example 2. Augmented main machine.

The result of the cross-product construction of two instances of the augmented main machine is shown in Figure 3.13. State vertex E^1F^2 indicates that the execute stage of the first instruction overlaps with the fetch of the second instruction. The edge from E^1F^2 to F^2 indicates that the execute stage of the first instruction finished before the fetch of the second instruction. The edge from E^1F^2 to E^2 indicates that the execute and fetch finished simultaneously. Note that the cross-product construction automatically captures all possible cases for a sequence of two operations.

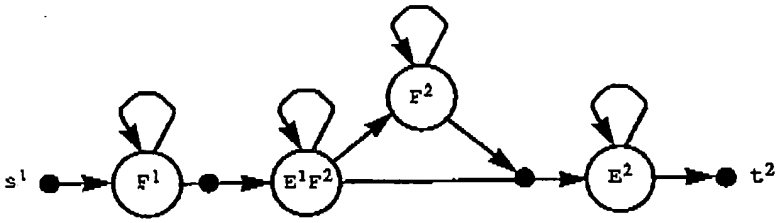


Figure 3.13: Example 2. Composition of machines M^1 and M^2.

This section has motivated the concept of composing main machines together to create execution sequences. As presented in this section, the machine M^* is an infinite machine obtained from the composition of infinite copies of the main machine. It is actually possible to obtain a closed form of the machine M^* that represents all possible execution sequences. An elegant and succinct representation for the main machine and a closed form of the machine M^* will be discussed later in Chapter 7.

3.3.3 Map Machines

The map machines relate abstract elements in the specification to node elements in the circuit. Each abstract input or output can be mapped into a protocol on a set of *action* and *reaction nodes*, as shown in Figure 3.14. The action nodes are in the same direction as the abstract signal. The

reaction nodes are in the reverse direction. Abstract states are mapped into a set of action nodes in the circuit.

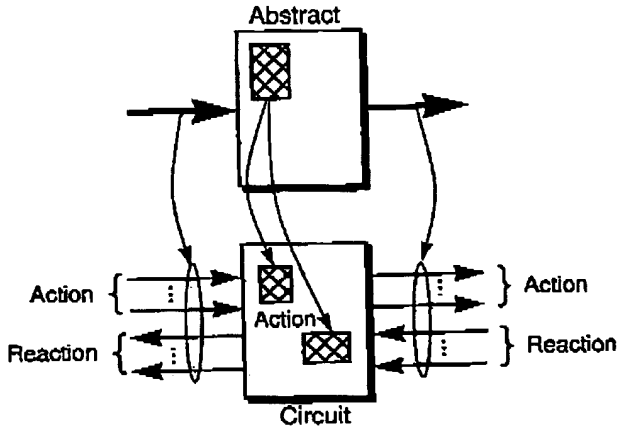


Figure 3.14: Role of action and reaction in the mapping.

The map machines define a spatial and temporal mapping for each simple abstract formula in the specification. A control graph is used to define the temporal aspect of the mapping. The spatial mapping is defined by node formulas on state vertices. Each state vertex in the control graph is associated with two node formulas, i.e., the *action* and *reaction node formulas*. The action node formulas are used to relate abstract inputs, outputs, and state elements to action nodes in the circuit. The reaction node formulas are used to relate abstract inputs and outputs to reaction nodes in the circuit. The map machines are not completely independent. A *synchronization function* maps event vertices in the map control graph to event vertices in the main control graph. The synchronization function relates the flow of control of the abstract element to the flow of control of the entire operation. The map machine can be viewed as a control graph with node formulas on state vertices and synchronization on event vertices as shown in Figure 3.15. Event vertices in the map machine are synchronized to event vertices in the main machine. Each state vertex is associated with node formulas. The action node formulas are shown in the upper half of the shadowed box and reaction node formulas are shown in the lower half of the shadowed boxes.

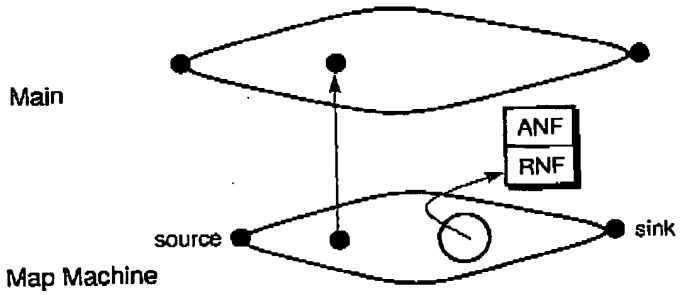


Figure 3.15: General form of the map machine.

As an example, assume an abstract ALU that is fetching an operand A from an abstract register file as shown in Figure 3.16. Consider a circuit in which the register file indicates that the operand is available with a rdyA signal and the ALU acknowledges receiving the operand with an ackA signal.

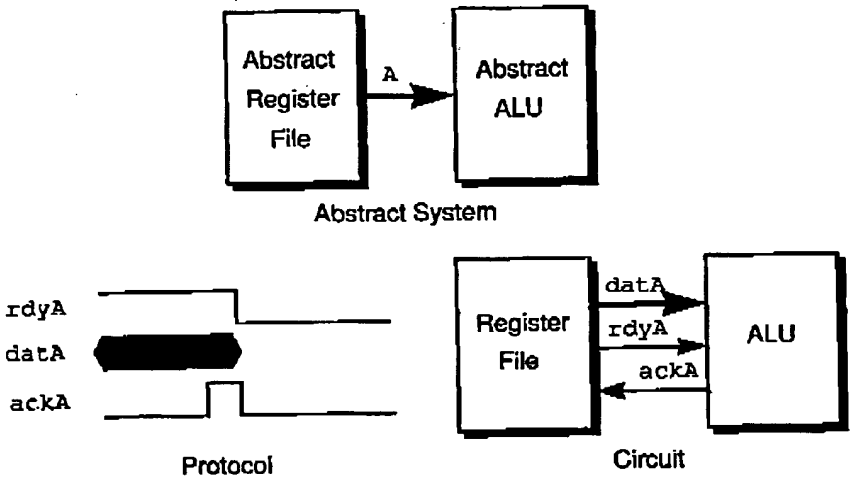


Figure 3.16: An abstract system and the corresponding circuit and protocol.

The signals datA and rdyA serve as action nodes since they are in the same direction as the abstract signal A. The signal ackA serves as the reaction node since it is in the reverse direction. Consider the simple abstract formula (A is 0) . The map machine for this simple abstract formula, written *Map*(A is 0) , is shown in Figure 3.17. The machine is defining the protocol shown in Figure 3.16, as a control graph with action and reaction node formulas. The action node formulas are shown in the upper half and the reaction node formulas in the lower half of the shadowed box.

The register file indicates that the operand is available by activating the ready signal in state vertex J. The register file has to wait for an arbitrary number of cycles before the ALU accepts the operand and activates the acknowledge signal in state vertex K. Finally, both the ready and acknowledge signals are de-activated in state vertex L. The synchronization function is specified so that the map machine is synchronized with the fetch stage of the main machine.

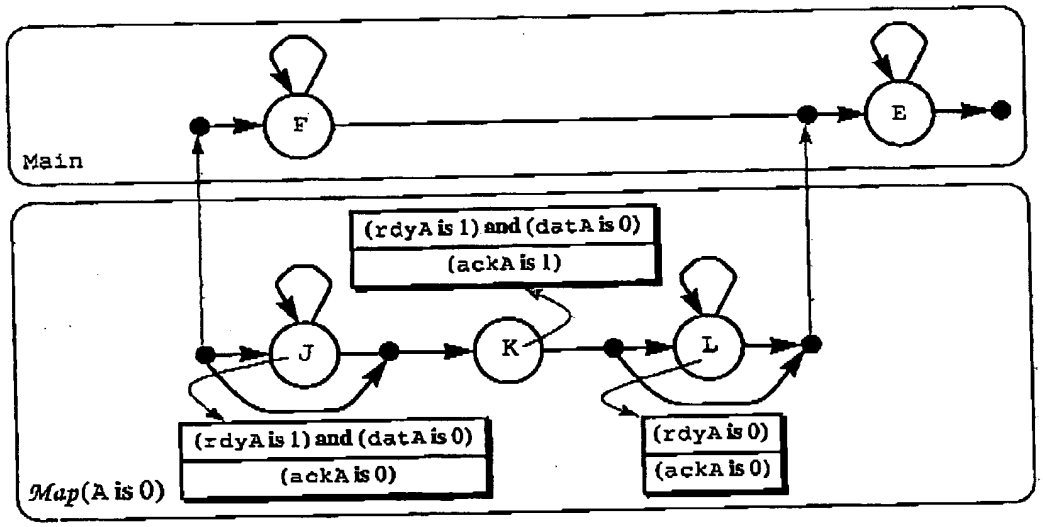


Figure 3.17: Map machine for (A is 0).

Later on, the map machines will be used to define the stimulus and response for a simulation based verification strategy. The action node formulas define the stimulus, and the reaction node formulas define the response for the simulator. Assume we are verifying the ALU in Figure 3.16. The abstract signal A serves as an input to the ALU and will appear in the precondition of the abstract assertions. The rdyA and datA signals serve as stimulus signals and ackA serves as the response signal for the ALU. Now, consider that we are verifying the register file. The abstract signal A serves as an output of the register file and will appear in the postcondition of the abstract assertions. A mirror operation is used to reverse the roles of the action and reaction formulas for abstract elements in the postcondition. The machine obtained from the mirror operation is shown in Figure 3.18. The ackA signal serves as the stimulus signal and the rdyA and datA signals serve as the response signals for the register file. This section has motivated the concept of using a single map machine to describe the interface protocols for interacting subsystems. A detailed

description of the mirror operation and how these map machines are used to verify subsystems will be discussed later in Chapter 4.

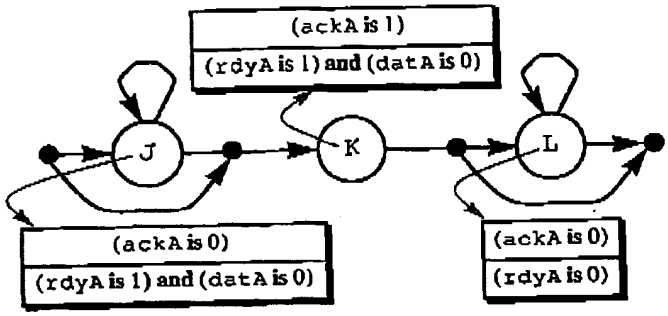


Figure 3.18: Mirror of map machine for (A is 0).

A mathematical definition of the map machine requires some terminology introduced in Section 2.3. For convenience, the relevant terminology is repeated here. The set S_v is the set of abstract elements in the abstract specification. Abstract elements are used to define abstract formulas. Abstract formulas are of the form: simple abstract formulas, conjunction, and domain restriction. Let $\mathcal{F}_{simple}$ represent the set of all simple abstract formulas. Each abstract element $s_i \in S_v$ is associated with two simple abstract formulas, i.e. $(s_i \text{ is } 0)$ and $(s_i \text{ is } 1)$. Therefore, if $n = |S_v|$, then $|\mathcal{F}_{simple}| = 2n$.

Each simple abstract formula is mapped into a map machine. The mapping is said to be complete if each simple abstract formula in $\mathcal{F}_{simple}$ is associated with a map machine. A complete mapping, therefore, requires $2n$ map machines. A map machine for a simple abstract formula $s \in \mathcal{F}_{simple}$ can be represented as a tuple of the form: $\mathcal{M}ap(s) = \langle V_s, U_s, E_s, s_s, t_s, \sigma_a, \sigma_r, \Upsilon_s \rangle$, where

- $\langle V_s, U_s, E_s, s_s, t_s \rangle$ is the control graph.
- σ_a is the labelling function that labels state vertices with action node formulas.
- σ_r is the labelling function that labels state vertices with reaction node formulas.
- Υ_s is the synchronization function, $\Upsilon_s : U_s \rightarrow U_m$. This is a partial function.

The map machines for abstract state elements will use only action node formulas. Also, abstract inputs and outputs that are mapped into unidirectional signals in the direction of the abstract signal will use only action node formulas. For these map machines, the reaction node formula is assumed

to be the trivially true node formula, i.e. $\forall(v_s \in V_s)$, $\sigma_r(v_s) = true$. We shall refer to these machines as *single-sided map machines*. Map machines that use both the action and reaction node formulas are referred to as *double-sided map machines*. Double-sided map machines are used to map abstract inputs and outputs into a protocol on a set of bidirectional signals in the circuit.

The synchronization function is a partial function. In a typical map machine, only a few event vertices will need to be synchronized to the main control graph. The rest of the event vertices will be *free* vertices in the sense that they are not required to be synchronized to a specific point in the main control graph. For a free vertex, $u_s \in U_s$, the synchronization function is defined to be $\Upsilon_s(u_s) = \Lambda$. Strictly speaking, the synchronization function should be of the form: $\Upsilon_s : U_s \rightarrow U_m \cup \Lambda$. For the sake of simplicity, however, we will use $\Upsilon_s : U_s \rightarrow U_m$ with the understanding that Λ represents a free vertex.

3.4 Extensions to Mapping Language

3.4.1 Symbolic Extension

The preceding section defined the scalar version of the mapping language. The language can be extended into the symbolic domain.

A *symbolic node formula* can define a set of scalar node formulas. A symbolic node formula is associated with a set of symbolic variables V, and is of the form:

- **Simple node formula**: $(n_i \text{ is } 0)$ and $(n_i \text{ is } 1)$ are symbolic node formulas if $n_i \in N_v$.
- **Conjunction**: $(F_1 \text{ and } F_2)$ is a symbolic node formula if F_1 and F_2 are symbolic node formulas.
- **Domain Restriction**: $(e \rightarrow F)$ is a symbolic node formula if F is a symbolic node formula and e is a Boolean expression over V.

Note that the only change from the definition of a scalar node formula is that the domain constraint can now be a Boolean expression rather than only 0 or 1.

We introduce the notation $(n_i \text{ is } e)$ as a shorthand for the formula $(\bar{e} \rightarrow (n_i \text{ is } 0))$ and $(e \rightarrow (n_i \text{ is } 1))$. That is, we constrain node element n_i to have the particular symbolic Boolean value denoted by the expression e.

In the basic form, each state element, s_i, was associated with two map machines, $\mathcal{M}ap(s_i \text{ is } 0)$ and $\mathcal{M}ap(s_i \text{ is } 1)$. A *symbolic map machine* can be used to capture both machines. A Boolean variable v_f is used to define a symbolic map machine, $\mathcal{M}ap(s_i \text{ is } v_f)$. The variable v_f serves as a formal argument that can be replaced by an arbitrary Boolean expression e to obtain the machine $\mathcal{M}ap(s_i \text{ is } e)$. The symbolic map machine uses symbolic node formulas defined over the set of symbolic variables $V = V_l \cup \{v_f\}$, where V_l is the set of local symbolic variables associated with the map machine.

As an example, a symbolic map machine $\mathcal{M}ap(\mathtt{A} \text{ is } \mathtt{a})$ can be used to define two scalar map machines, $\mathcal{M}ap(\mathtt{A} \text{ is } 0)$ and $\mathcal{M}ap(\mathtt{A} \text{ is } 1)$. The variable a serves as the formal argument.

3.4.2 Vector and Data Handling Extensions

So far, the node elements have been limited to single bit and the symbolic variables have been limited to Boolean variables. Data handling extensions can be used to ease the task of writing the mapping. The user can define an array of nodes and use symbolic indexing to index into these arrays. The array of nodes can be aliased into actual net names in the realization. The user can define complex symbolic variables. Expressions over symbolic variables can be defined using logical, bitwise, arithmetic, and relational operators.

As an example, consider the abstract assertion for a bitwise-OR operation introduced in Section 2.4.2. The mapping language requires three map machines, i.e., $\mathcal{M}ap(\mathtt{SA} \text{ is } \mathtt{u})$, $\mathcal{M}ap(\mathtt{SB} \text{ is } \mathtt{v})$ and $\mathcal{M}ap(\mathtt{ST} \text{ is } \mathtt{w})$, where u, v, and w are 32-bit symbolic variables that serve as formal arguments for state elements SA, SB, and ST respectively. Note that each of these symbolic machines captures 2^{32} scalar map machines corresponding to all possible assignments to the symbolic variables.

3.5 Syntax and Examples

The syntax of the mapping language is described in Appendix B. This section discusses some examples.

3.5.1 Addressable Accumulator

The abstract specification for the addressable accumulator was shown in Figure 2.9. Let us consider a circuit with the timing of the add operation as shown in Figure 3.19. The circuit uses a 4-phase non-overlapping clocking scheme. The `Clear` signal defines the add or store operation.

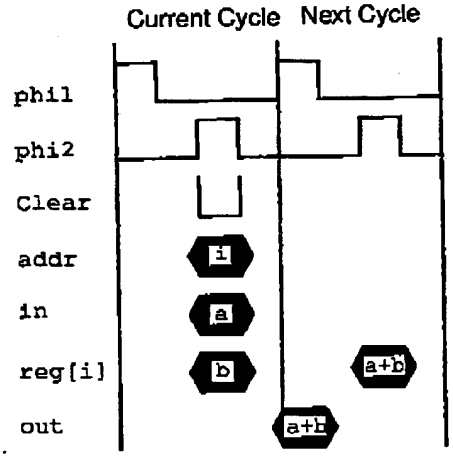


Figure 3.19: Timing for add operation in the accumulator.

The mapping for such a circuit, shown below, can be divided into 5 separate sections: 1. Node declarations 2. Alias definitions 3. Hierarchical state vertices 4. Main machine and 5. Map machines. We will discuss each of these sections in detail.

```
MACHINE accumulator;
    Node Declarations
    Alias Definitions
    Hierarchical State Vertices
    Main Machine
    Map Machines
ENDMACHINE
```

The node declaration section is shown below. This section defines all the nodes used in the mapping.

```
NODE phi1, phi2, Clear;
NODE reg[31 to 0][15 to 0];
NODE addr[3 to 0];
NODE in[15 to 0], out[15 to 0];
```

The alias definition section is shown below. The aliases map node elements into actual net names in the realization. As an example the node element `addr[15 TO 0]` is mapped into nets `Addr.15, Addr.14 ... Addr.0`.

```
ALIAS reg[x][y] {Reg.}[x]{.}[y];
ALIAS addr[x] {Addr.}[x];
ALIAS in[x] {In.}[x];
ALIAS out[x] {Out.}[x];
```

The hierarchical state vertices section is shown below. Each hierarchical level is defined as an `ENTITY` in the mapping. The entity in the figure is defining the 4 phase non-overlapping clocking scheme shown in Figure 3.5. The vertex declarations and next definition define the control graph. The control graph is defined in terms of phase-level state vertices. The label definition defines the node formulas on the state vertices.

```
ENTITY cycle {
    VERTEX P1, P2, P3, P4: PHASE;
    VERTEX start, u, v, w, end: EVENT;
    NEXT {
        start: P1;
        P1: u;
        u: P2;
        P2: v;
        v: P3;
        P3: w;
        w: P4;
        P4: end;
    }
    LABEL {
        (phi1 IS '1' at P1) and (phi2 IS '0' at P1) and
        (phi1 IS '0' at P2) and (phi2 IS '0' at P2) and
        (phi1 IS '0' at P3) and (phi2 IS '1' at P3) and
        (phi1 IS '0' at P4) and (phi2 IS '0' at P4)
    }
}
```

The main machine, shown below, is defined as the top-level entity. The control graph defines the flow of control for store and add operations. The nextmarker defines the time point where the next

operation can start. A pictorial view of the main machine is shown in the upper half of Figure 3.20. Each state vertex is a cycle-level state vertex consisting of 4 phases.

```
ENTITY main {
    VERTEX CurrCyc, NextCyc: cycle;
    VERTEX: currStart, nextStart, done: EVENT;
    NEXT {
        currStart: CurrCyc;
        CurrCyc: nextStart;
        nextStart: NextCyc;
        NextCyc: done;
    }
    NEXTMARKER {currStart: nextStart};
}
```

The single-sided map machine for the abstract element op is shown below. The synchronization function maps the start of the map machine to the start of the main machine. The labelling defines the action node formulas. The reaction node formula is assumed to be the trivially true node formula. A pictorial view of the map machine is shown in the lower half of Figure 3.20.

```
MAP (op IS o) TO {
    VERTEX Cyc: cycle;
    VERTEX start, end: EVENT;
    NEXT {
        start: Cyc;
        Cyc: end;
    }
    SYNCH {start: main.currStart};
    CASE (o) {
        IS store:
            LABEL {Clear is '1' at Cyc.P3}
        IS add:
            LABEL {Clear is '0' at Cyc.P3}
    }
}
```

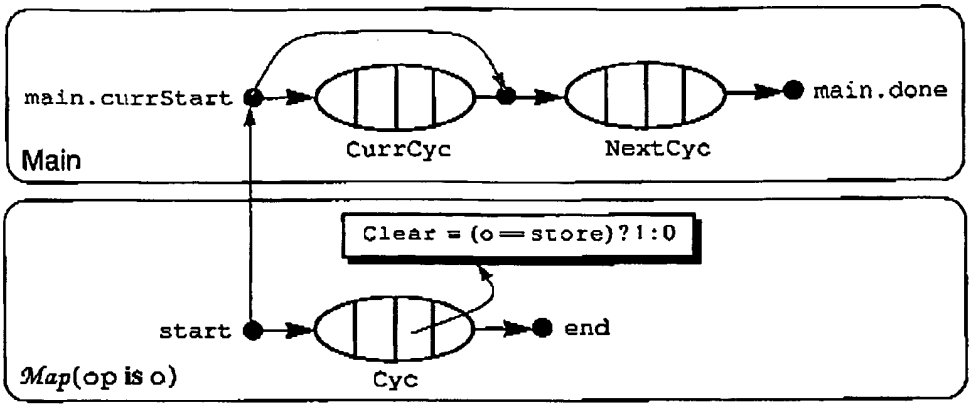


Figure 3.20: Pictorial view of the main and map machine for the accumulator.

The rest of the map machines are similar and are shown below.

```
MAP (register[i] IS r) TO {
    VERTEX Cyc: cycle;
    VERTEX start, end: EVENT;
    NEXT {start: Cyc; Cyc: end;}
    SYNCH {start: main.currStart};
    LABEL {reg[i] is r at Cyc.P3}
}

MAP (address IS u) TO {
    VERTEX Cyc: cycle;
    VERTEX start, end: EVENT;
    NEXT {start: Cyc; Cyc: end;}
    SYNCH {start: main.currStart};
    LABEL {addr is u at Cyc.P3}
}

MAP (dataIn IS d) TO {
    VERTEX Cyc: cycle;
    VERTEX start, end: EVENT;
    NEXT {start: Cyc; Cyc: end;}
    SYNCH {start: main.currStart};
    LABEL {In is d at Cyc.P3}
}

MAP (dataOut IS d) TO {
    VERTEX Cyc: cycle;
    VERTEX start, end: EVENT;
    NEXT {start: Cyc; Cyc: end;}
    SYNCH {start: main.currStart};
    LABEL {out is d at Cyc.P1}
}
```

The abstract elements op, address, and dataIn are abstract inputs that will appear only in the precondition. The node formulas in the map machines for these abstract inputs define the stimulus for the circuit. The abstract element dataOut is an abstract output that will appear only in the postcondition. The node formulas in the map machine for dataOut define the desired response from the circuit. The abstract element register[i] is an abstract state that can appear both in the precondition and postcondition. In the precondition, the node formulas in the map machine for the abstract register define constraints on internal state elements in the circuit. In the postcondition, the node formulas in the same map machine define the desired state transitions for the circuit.

3.5.2 Pipelined Addressable Accumulator

Let us consider another realization of the addressable accumulator shown in Figure 3.21. The pipeline register Hold has been incorporated to achieve a greater throughput by overlapping the adder and register write operations. While the adder is performing the sum for the current cycle, the value from the previous cycle is written into the register array. If the previous address in the control logic is the same as the current address, then the accumulator bypasses the register array and transfers the data directly from the Hold register to the input of the adder.

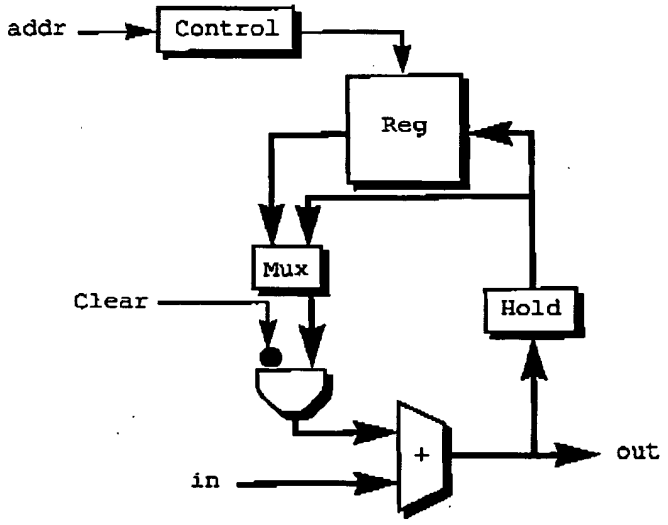


Figure 3.21: A pipelined addressable accumulator realization.

The timing for the add operation in this pipelined addressable accumulator is shown in Figure 3.22. Note that if the previous address k is the same as the current address i, then the adder takes in the data from the Hold register. Else, if the two addresses are different, then the data is taken from the register file.

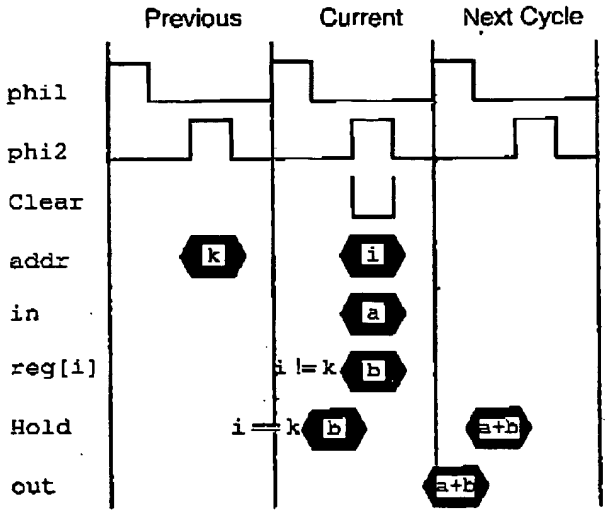


Figure 3.22: Timing for the add operation in the pipelined accumulator.

The main machine for the pipelined accumulator is shown below. The main machine has 3 cycle-level state vertices corresponding to the previous, current, and next cycle.

```
ENTITY main {
    VERTEX PrevCyc, CurrCyc, NextCyc: cycle;
    VERTEX prevStart, currStart, nextStart, done: EVENT;
    NEXT {
        prevStart: PrevCyc;
        PrevCyc: currStart;
        currStart: CurrCyc;
        CurrCyc: nextStart;
        nextStart: NextCyc;
        NextCyc: done;
    }
    NEXTMARKER {prevStart: currStart};
}
```

The pipelined accumulator compares the current address with the previous address. The previous address is defined in an append assertion section as shown below. Note that the address k is synchronized with the start of the main machine. The start of the main machine corresponds to the previous cycle.

```
APPEND AddAssertion {
    VARIABLE k: ADDRWORD;
    (address IS k SYNCH main.prevStart) LEADSTO ()
}
```

Most of the map machines for the pipelined accumulator are the same as for the nonpipelined version. The map machine for the register file, however, is a little more complicated and is shown below. This is a *state dependent mapping* since it is dependent on the previous address. The COND keyword is used to define state dependent mappings. The abstract register is mapped into the `Hold` register if the previous address `k` is the same as the current address `i`. Else the abstract register is mapped into the register file `reg`.

```
MAP (register[i] IS r) TO {
    VERTEX Cyc: cycle;
    VERTEX start, end: EVENT;
    NEXT {start: Cyc; Cyc: end}
    SYNCH {start: main.currStart};
    COND (address IS k SYNCH main.prevStart) {
        WHEN (i == k) {
            LABEL {Hold is r at Cyc.P2}
        } ELSE {
            LABEL {reg[i] is r at Cyc.P3}
        }
    }
}
```

Consider the precondition of the add assertion in Figure 2.9. The append assertion section defined the previous address to be the symbolic value `k`. The abstract formula `(register[i] is b)` in the precondition will be mapped into a constraint on the pipeline hold register and the register array in the circuit. The hold register will be constrained to the value `b` under the condition $i == k$, and the register array will be constrained to the value `b` under the condition $i \neq k$. Now, consider the postcondition of the add assertion. The previous address in the postcondition is the symbolic address `i`. The abstract formula `(register[i] is a + b)` in the postcondition will be mapped into a check for the value `a + b` in the hold register.

3.6 Miscellaneous Issues

One of the concerns in this work is that the implementation mapping can become very complex. An area of focus for future work would be to simplify or automate the generation of the mapping information as much as possible. Another possibility is to explore notations such as annotated timing diagrams for expressing the mapping. Formalisms on timing diagrams have been studied earlier[85][86]. Some of the concepts might be applicable to this problem.

Chapter 4

Trajectory Generation

The previous two chapters have described the form of the abstract specification and the implementation mapping. The abstract specification defines each operation of the system as an abstract assertion. Each abstract assertion is defined in the form of a precondition leads to a postcondition. The precondition and postcondition are abstract formulas that operate on abstract elements. The implementation mapping provides a temporal and spatial mapping from the set of abstract elements to the set of node elements in the circuit.

The trajectory generation phase takes the abstract specification and mapping and generates the *trajectory specification*. The trajectory specification consists of a set of *trajectory assertions*. Each abstract assertion gets mapped into a trajectory assertion. Each abstract formula is mapped into a *trajectory formula*. The trajectory assertion is a control graph that captures all possible nondeterministic cases for the circuit. We use the terms trajectory specification and trajectory assertion partly for historical reasons. Our trajectory assertions are a generalization of the trajectory assertions introduced by Seger and Bryant[18]. The justification is that a trajectory assertion defines a set of trajectories in the simulator. Informally, a trajectory can be considered to be a sequence of states that represents an acceptable behavior of the circuit. A rigorous definition of a trajectory will be given later in Chapter 6.

4.1 Related Work

Seger and Bryant introduced the terms trajectory formula and trajectory assertion[18]. The trajectory formula used the next-time operator to define restrictions on circuit nodes for a finite duration of time. In effect, their trajectory formula was a single sequence of state vertices labelled with action node formulas. Seger and Bryant described a simple trajectory assertion in the form of an implication, i.e. the antecedent implies the consequent. They used the sequence and iteration constructs to create more complex trajectory assertions. In effect, their simple trajectory assertion was

a single sequence of state vertices labelled with action and reaction node formulas. The iteration construct augmented the single sequence with a limited set of loops. In particular, they did not allow nested or interacting loops. Our trajectory formulas and trajectory assertions are arbitrary control graphs with state vertices labelled with both action and reaction node formulas.

Beatty used the abstract specification and implementation mapping to generate a trajectory specification[13]. The mapping, however, could express only a limited set of temporal behaviors. Beatty described the mapping in terms of a formulation called marked strings that could express only bounded single behavior sequences. Abstract inputs and outputs could only be mapped into a set of unidirectional signals in the circuit. In effect, the mapping was limited to single sequence of state vertices labelled with action node formulas. Beatty defined an overlapped concatenation operator over marked strings that was used to generate the trajectory assertion. The resultant trajectory assertion was a single sequence of state vertices.

Our map machines are arbitrary control graphs with both action and reaction node formulas. Reaction node formulas are used to map abstract inputs and outputs into protocols on bidirectional signals. The trajectory generation phase essentially performs a cross-product construction of the control graphs. The resultant trajectory assertion is a graph labelled with both action and reaction node formulas.

The next section gives a brief overview of trajectory generation. A detailed description is given in subsequent sections.

4.2 Overview of Trajectory Generation

An abstract formula gets mapped into a trajectory formula. The trajectory formula is the set of the trajectories defined by the abstract formula for a particular implementation. For the moment, consider a trajectory formula to be a control graph with node formulas on state vertices and a synchronization function. Later on, we will give a more rigorous mathematical definition of a trajectory formula. Notice that a trajectory formula has the same form as a map machine. The map machines can serve as trajectory formulas for the simple abstract formulas. Given the map machines, the tra-

jectory formula for an abstract formula AF, written $\mathcal{TF}(AF)$, can be automatically computed as follows:

- **Simple Abstract Formulas:** $\mathcal{TF}(s_i \text{ is } v) = \mathcal{M}ap(s_i \text{ is } v)$, where $v \in \{0, 1\}$.
- **Conjunction:** $\mathcal{TF}(AF_1 \text{ and } AF_2) = \mathcal{TF}(AF_1) \,||\, \mathcal{TF}(AF_2)$, where $||$ denotes the *compose* operator. Composition amounts to taking the cross-product of the two control graphs under restrictions specified by the synchronization function. Assume that v_1 and v_2 are state vertices in the trajectory formulas for AF_1 and AF_2 respectively. Further, assume that ANF_1 and RNF_1 are the action and reaction node formulas associated with state vertex v_1. And ANF_2 and RNF_2 are the action and reaction node formulas associated with state vertex v_2. Then, the action and reaction node formulas associated with the vertex $\langle v_1, v_2 \rangle$ in the cross-product construction are $(ANF_1 \text{ and } ANF_2)$ and $(RNF_1 \text{ and } RNF_2)$ respectively. This is shown pictorially in Figure 4.1. Intuitively, the cross-product construction captures all possible interactions between the two machines.

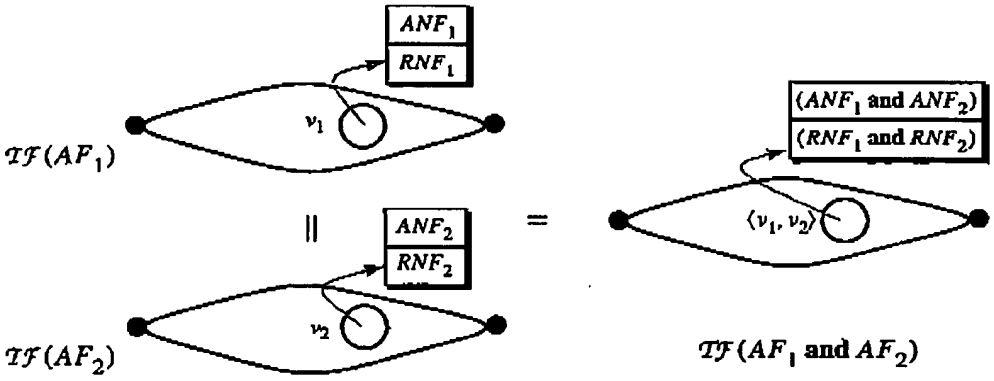
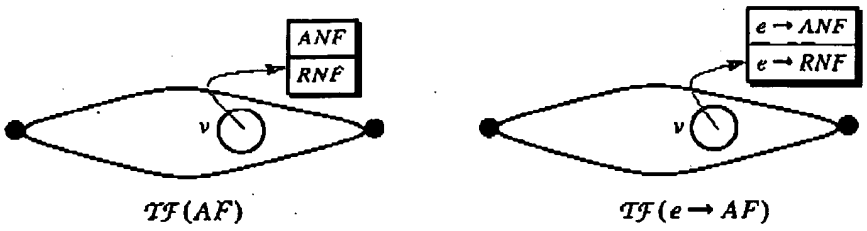


Figure 4.1: Trajectory formula for conjunction.

- **Domain Restriction:** $\mathcal{TF}(e \rightarrow AF) = \mathcal{TF}(AF)|_{\sigma/e \rightarrow \sigma}$, where $\sigma/e \rightarrow \sigma$ denotes the fact that σ_a and σ_r are replaced by $e \rightarrow \sigma_a$ and $e \rightarrow \sigma_r$ respectively. Assume that v is a state vertex in the trajectory formula for AF, with action node formula ANF and reaction node formula RNF. The action and reaction node formulas associated with the vertex v for the trajectory formula $(e \rightarrow AF)$ are $(e \rightarrow ANF)$ and $(e \rightarrow RNF)$ respectively. This is shown pictorially in Figure 4.2. Intuitively, the domain restriction is specifying that the abstract formula AF has

to hold only when the expression e evaluates to true. Therefore, the action and reaction node formulas have to define a set of stimuli and responses for the simulator only when the expression evaluates to true.



$\mathcal{TF}(AF)$ $\mathcal{TF}(e \rightarrow AF)$

Figure 4.2: Trajectory formula for domain restriction.

For an abstract assertion $A = P \Rightarrow Q$, $\mathcal{TF}(P)$ and $\mathcal{TF}(Q)$ are the trajectory formulas associated with the precondition and postcondition respectively. The trajectory assertion corresponding to A can be automatically computed as $\mathcal{TA}(A) = \mathcal{TF}(P) // [\mathcal{TF}(Q)]^M$, where // is the shift-and-compose operation and the superscript ^{M} is the mirror operation. The mirror operation reverses the role of the action and reaction node formulas. The shift-and-compose operator shifts the start of the machine $\mathcal{TF}(Q)$ to the nextmarker of the source in the main machine and then performs the composition. This is shown pictorially in Figure 4.3. Assume that v_1 and v_2 are state vertices in the trajectory formula for the precondition and postcondition respectively. Further assume that ANF_1 and RNF_1 are the action and reaction node formulas associated with vertex v_1. Similarly let ANF_2 and RNF_2 be the action and reaction node formulas associated with vertex v_2. The mirror operation reverses the role of the node formulas ANF_2 and RNF_2 in the postcondition. Consider the vertex $\langle v_1, v_2 \rangle$ in $\mathcal{TA}(P \Rightarrow Q)$. The action and reaction node formula associated with vertex $\langle v_1, v_2 \rangle$ are (ANF_1 and RNF_2) and (RNF_1 and ANF_2) respectively.

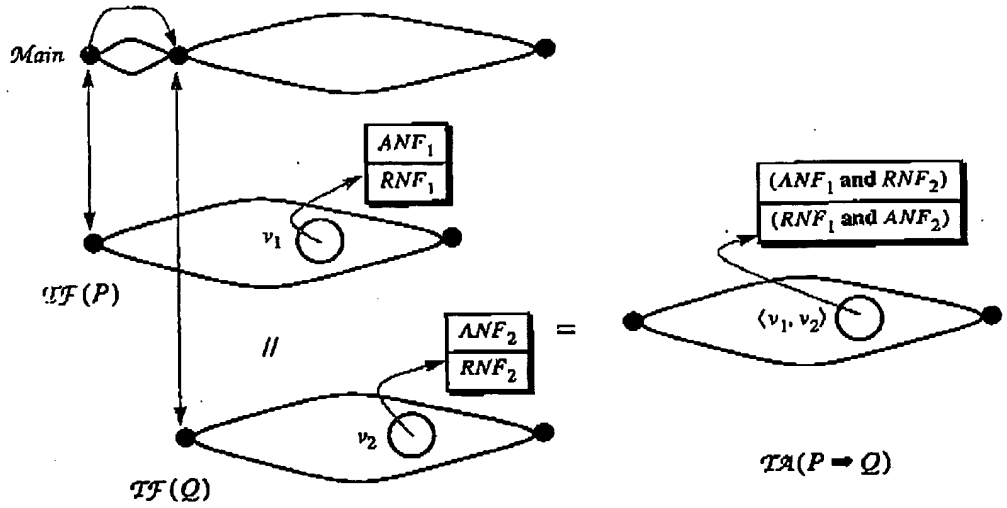


Figure 4.3: Generating the trajectory assertion.

In the precondition, the action node formula ANF_1 refers to circuit inputs and state elements that define the stimulus and current state for the circuit. The reaction node formula RNF_1 refers to circuit outputs that define the desired response from the circuit. In the postcondition, however, the action node formula ANF_2 refers to circuit outputs and state elements that define the desired response and state transitions. The reaction node formula RNF_2 refers to circuit inputs that define the stimulus for the circuit. In the trajectory assertion $\mathcal{TA}(P \Rightarrow Q)$, the action node formulas define the stimuli and current state for the circuit. Action node formulas are used as generators since they generate low-level signals for the circuit. The reaction node formulas define the desired response and state transitions. Reaction node formulas are used as acceptors since they define the set of acceptable responses from the circuit.

Incidentally the shift-and-compose operator was also used to stitch operations together to form execution sequences in Section 3.3.2, with the slight variation that the main machines were not associated with any node formulas.

The above overview defines the basic concepts of trajectory generation. The actual details are a bit more complicated. The main reason for the added complication is that the map machines are not synchronized with each other but are all synchronized to the main machine. This requires the main

machine to be brought into the composition process. The next few sections deal with the actual details of trajectory generation.

4.3 Trajectory Formula

Let us now rigorously define a trajectory formula. A trajectory formula for an abstract formula AF is defined to be a tuple of the form: $\mathcal{TF}(AF) = \langle V_t, U_t, E_t, s_t, t_t, \sigma_a, \sigma_r, \Upsilon_t \rangle$, where

- $\langle V_t, U_t, E_t, s_t, t_t \rangle$ is a control graph.
- σ_a is a labelling function that labels state vertices with action node formulas.
- σ_r is a labelling function that labels state vertices with reaction node formulas.
- Υ_t is a projection function that relates event vertices in the trajectory formula to event vertices in the main machine, $\Upsilon_t : U_t \rightarrow U_m$. This is a total function, i.e., Υ_t relates every event vertex in the trajectory formula to an event vertex in the main machine.

The previous section presented a simplified overview of trajectory generation. Unfortunately, the map machine cannot serve as the trajectory formula for simple abstract formulas. The reason is that the map machines are synchronized to the main machine. The main machine has to be brought in to the definition of the trajectory formula. Let $\mathcal{Main} = \langle V_m, U_m, E_m, s_m, t_m, \beta_m \rangle$ represent the main machine. A rigorous mathematical definition of the trajectory formula requires two forms of the compose operator. We shall refer to these as the *dot-composition* and the *parallel-composition* operators. The symbol • will be used to denote the dot-composition operator and the symbol || will be used to denote the parallel-composition operator. Given these operators, the trajectory formula can be recursively defined as:

- **Simple Abstract Formulas:** $\mathcal{TF}(s_i \text{ is } v) = \mathcal{Main} \bullet \mathcal{Map}(s_i \text{ is } v)$.
- **Conjunction:** $\mathcal{TF}(AF_1 \text{ and } AF_2) = \mathcal{TF}(AF_1) \,||\, \mathcal{TF}(AF_2)$.
- **Domain Restriction:** $\mathcal{TF}(e \rightarrow AF) = \langle V_t, U_t, E_t, s_t, t_t, e \rightarrow \sigma_a, e \rightarrow \sigma_r, \Upsilon_t \rangle$.

Most of the complexity due to synchronization is captured in the dot-composition operator. The next few sections describe the composition operators in detail.

4.4 Dot-Composition

A high-level view of the dot-composition process is shown in Figure 4.4. We are given the main machine and the map machine $\mathcal{M}ap(s_i \text{ is } v)$. Some of the event vertices in the map machine are synchronized to event vertices in the main machine. The dot-composition process will result in the trajectory formula $\mathcal{TF}(s_i \text{ is } v)$. All the event vertices in the trajectory formula will be related to event vertices in the main machine.

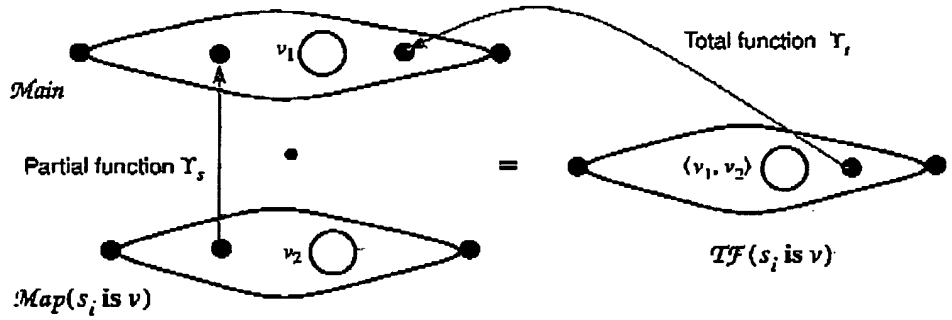


Figure 4.4: High-level view of the dot-composition process.

There are three separate steps involved in the dot-composition $\mathcal{M}ain \bullet \mathcal{M}ap(s_i \text{ is } v)$:

- Augment the map machine $\mathcal{M}ap(s_i \text{ is } v)$.
- Precompose $\mathcal{M}ain$ and augmented $\mathcal{M}ap(s_i \text{ is } v)$ |
- Prune the graph obtained from the precomposition step to generate $\mathcal{M}ain \bullet \mathcal{M}ap(s_i \text{ is } v)$.

4.4.1 Augment Map Machine

The precompose step requires that the source of the map machine is synchronized with the source of the main machine and the sink of the map machine is synchronized with the sink of the main machine. In general this might not be the case. The purpose of this step is to augment the map machine with dummy vertices to ensure this condition.

Let us consider a map machine of the form: $\mathcal{M}ap(s_i \text{ is } v) = \langle V_s, U_s, E_s, s_s, \iota_s, \sigma_a, \sigma_r, \Upsilon_s \rangle$. Let us first consider the case where the source of the map machine is not synchronized with the source of the main machine, i.e. $\Upsilon_s(s_s) \neq s_m$. Then, the map machine is augmented with a dummy state

vertex v_a and a new source vertex s_a as shown in Figure 4.5. The new source vertex is synchronized with the source of the main machine i.e. $\Upsilon_s(s_a) = s_m$. The dummy state vertex is associated with the trivially true node formulas i.e. $\sigma_a(v_a) = true$ and $\sigma_r(v_a) = true$.

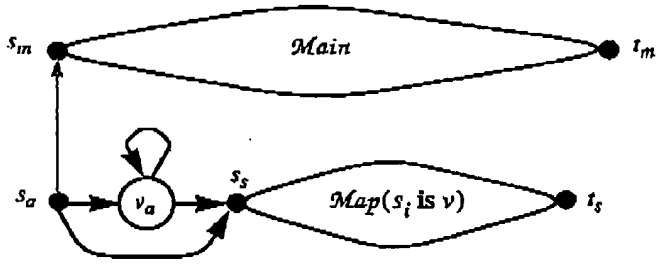


Figure 4.5: Augmenting source of map machine.

Let us now consider the case where the sink of the map machine is not synchronized with the sink of the main machine i.e. $\Upsilon_s(t_s) \neq t_m$. Then the map machine is augmented with a dummy vertex v_b and a new sink vertex t_a as shown in Figure 4.6. The new sink vertex is synchronized with the sink of the main machine i.e. $\Upsilon_s(t_a) = t_m$. The dummy state vertex is associated with the trivially true node formulas i.e. $\sigma_a(v_b) = true$ and $\sigma_r(v_b) = true$.

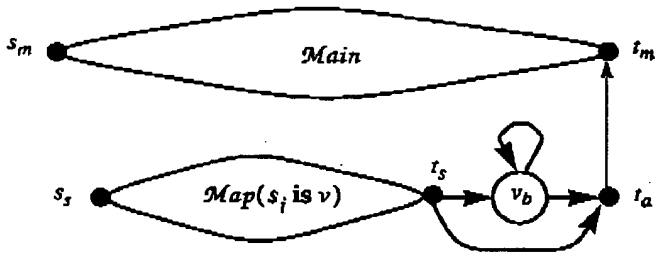


Figure 4.6: Augmenting sink of map machine.

4.4.2 Precomposition

A detailed description of the precomposition operator requires some additional mathematical notation. Let us consider a control graph of the form: $\langle V, U, E, s, t \rangle$. An *instantaneous path* is defined to be $p = w_0, w_1, w_2, \ldots, w_{n+1}$, such that $w_i \in U$, $\forall(i = 1 \ldots n)$ and $(w_i, w_{i+1}) \in E$, $\forall(i = 0 \ldots n)$. In other words, an instantaneous path is a path in the control graph where every vertex except the endpoint vertices have to be event vertices. The endpoint vertices can be either state or event vertices. Define the functions, *head*, *suffix*, and *midfix* of a path as $head(p) = w_0$,

and $suffix(p) = w_1, w_2, \ldots, w_{n+1}$, and $midfix(p) = w_1, w_2, \ldots, w_n$. In the degenerate case, an instantaneous path can consist of a single vertex i.e. $p = w_0$. In this case $head(p) = w_0$, and $suffix(p) = \Lambda$, and $midfix(p) = \Lambda$.

Let $\langle V_m, U_m, E_m, s_m, t_m, \beta_m \rangle$ represent the main machine and let $\langle V_s, U_s, E_s, s_s, t_s, \sigma_a, \sigma_r, \Upsilon_s \rangle$ represent the *augmented* map machine. Since this is an augmented map machine, the source is synchronized with the source of the main machine i.e. $\Upsilon_s(s_s) = s_m$. Also the sink is synchronized with the sink of the main machine i.e. $\Upsilon_s(t_s) = t_m$. Pre-composition is the cross-product of the two control graphs under restrictions specified by the synchronization function. Pre-composition results in a trajectory formula of the form: $\langle V, U, E, s, t, \sigma_a, \sigma_r, \Upsilon \rangle$, where

- V is the set of state vertices, $V \subseteq V_m \times V_s$.
- U is the set of event vertices[1], $U \subseteq (U_m \times U_s) \cup (U_m \times V_s) \cup (V_m \times U_s)$.
- E is the set of edges.
- s is the source, $s = \langle s_m, s_s \rangle$.
- t is the sink, $t = \langle t_m, t_s \rangle$.
- σ_a labels state vertices with action node formulas. For state vertices $v_m \in V_m$ and $v_s \in V_s$, and the state vertex $\langle v_m, v_s \rangle \in V$, the associated action node formula is $\sigma_a(\langle v_m, v_s \rangle) = \sigma_a(v_s)$.
- σ_r labels state vertices with reaction node formulas. For state vertices $v_m \in V_m$ and $v_s \in V_s$, and the state vertex $\langle v_m, v_s \rangle \in V$, the associated reaction node formula is $\sigma_r(\langle v_m, v_s \rangle) = \sigma_r(v_s)$.
- Υ relates event vertices to event vertices in the main machine. For vertices $u_m \in U_m$ and $w_s \in V_s \cup U_s$, and the event vertex $\langle u_m, w_s \rangle \in U$, the associated projection function is $\Upsilon(\langle u_m, w_s \rangle) = u_m$.

The pseudo code for the precomposition algorithm is shown in Figure 4.7. The routine is called as **preDotCompose**(s_m, s_s). Line 3 creates all the state vertices. For these state vertices, the node

1. A careful analysis of the preDotCompose algorithm given later in Figure 4.7 will show that actually $U \subseteq (U_m \times \{s_s\}) \cup (U_m \times V_s) \cup (\{t_m\} \times \{t_s\})$.

formulas are set to be $\sigma_a(\langle a_0, b_0\rangle) = \sigma_a(b_0)$ and $\sigma_r(\langle a_0, b_0\rangle) = \sigma_r(b_0)$. Line 3 also creates the source event vertex $\langle s_m, s_s\rangle$ and the sink event vertex $\langle t_m, t_s\rangle$. These two event vertices are related to the source and sink of the main machine. The rest of the event vertices are created in line 8. These event vertices are projected to the main machine so that $\Upsilon(\langle a_i, b_0\rangle) = a_i$. The algorithm is defined in terms of instantaneous paths. The path $p_m = a_0, a_1, \ldots, a_x, a_{x+1}$ is an instantaneous path in the main machine such that the head of a path is the source or a state vertex, i.e. $a_0 \in \{s_m \cup V_m\}$, and the tail of the path is the sink or a state vertex, i.e. $a_{x+1} \in \{V_m \cup t_m\}$. Similarly, the path $p_s = b_0, b_1, \ldots, b_y, b_{y+1}$ is an instantaneous path in the augmented map machine such that $b_0 \in \{s_s \cup V_s\}$ and $b_{y+1} \in \{V_s \cup t_s\}$. Given paths p_m and p_s, the precomposition will create an instantaneous path $\langle a_0, b_0\rangle, \langle a_1, b_0\rangle, \ldots, \langle a_x, b_0\rangle, \langle a_{x+1}, b_{y+1}\rangle$ in the resultant trajectory formula iff paths $midfix(p_m)$ and $midfix(p_s)$ satisfy the *path synchronization property*. The property takes care of the restrictions placed by the synchronization function.

```
vertex preDotCompose(a_0, b_0) {
    if ⟨a_0, b_0⟩ not in hash table {
        Create ⟨a_0, b_0⟩ and put in hash table.                    ..... Line 3
        foreach instantaneous path p_m = a_0, a_1, ..., a_{x+1} such that a_{x+1} ∈ V_m ∪ t_m {
            foreach instantaneous path p_s = b_0, b_1, ..., b_{y+1} such that b_{y+1} ∈ V_s ∪ t_s {
                if pathSynchProperty( midfix(p_m), midfix(p_s) ) {
                    ⟨a_{x+1}, b_{y+1}⟩ = preDotCompose(a_{x+1}, b_{y+1})
                    Create event vertices ⟨a_i, b_0⟩  ∀(i = 1...x)      ..... Line 8
                    Create path ⟨a_0, b_0⟩, ⟨a_1, b_0⟩, ..., ⟨a_x, b_0⟩, ⟨a_{x+1}, b_{y+1}⟩
                }
            }
        }
    }
    return ⟨a_0, b_0⟩
}
bool pathSynchProperty(pe_m, pe_s) {
    if (pe_s == Λ) return (true)
    if (pe_m == Λ) return (false)
    if ( Υ_s(head(pe_s)) == Λ ) return pathSynchProperty(pe_m, suffix(pe_s))
    if ( Υ_s(head(pe_s)) == head(pe_m) ) return pathSynchProperty(pe_m, suffix(pe_s))
    return pathSynchProperty( suffix(pe_m), pe_s)
}
```

Figure 4.7: Pseudo code for the precomposition algorithm.

The path synchronization property takes instantaneous paths and returns a Boolean value. The instantaneous paths pe_m and pe_s consist of only event vertices. Let us consider a path $a_0, \ldots, a_{x+1}$ in the main machine and a path $b_0, \ldots, b_{y+1}$ in the map machine as shown in Figure 4.8. There are two possible conditions for the path $b_1, \ldots, b_y$:

1. Every event vertex in the path is a free vertex, i.e. $\Upsilon_s(b_i) = \Lambda$, $\forall i$: In this case the precomposition algorithm will go ahead and create the ?-path.

2. There is at least one event vertex in the path that is synchronized to the main machine: Let us assume that b_j is the first non-free event vertex in the path. Therefore, $\Upsilon_s(b_j)$ is an event vertex in the main machine. Now there are two possible conditions for the path $a_1, \ldots, a_x$:

2a. The event vertex $\Upsilon_s(b_j)$ exists in the path: Let us assume that a_k is that vertex, i.e., $a_k = \Upsilon_s(b_j)$. This means that the synchronization condition for the vertex b_j has been met. Now recursively check the synchronization property for the paths $a_k, \ldots, a_x$ and $b_{j+1}, \ldots, b_y$

2b. The event vertex $\Upsilon_s(b_j)$ does not exist in the path: This means that the synchronization condition for the vertex b_j cannot be met. In this case the precomposition algorithm will not create the ?-path.

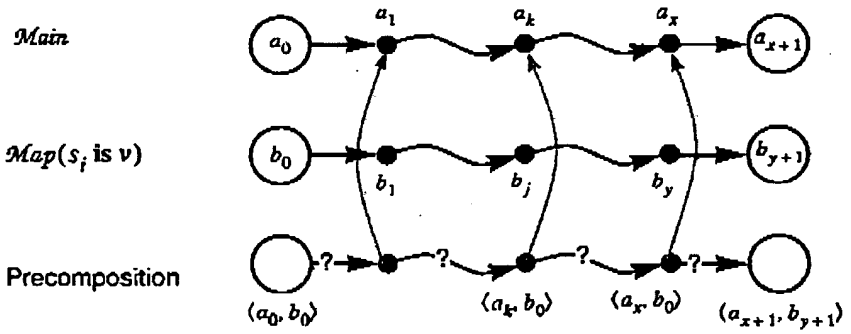


Figure 4.8: The path synchronization property.

Consider the example shown in Figure 4.9. The source of the map machine is synchronized to the source of the main machine. The user specified the map machine such that the event vertex b4 was synchronized to the event vertex a2 in the map machine. The map machine was augmented with the dummy state vertex DB4 and the event vertex db5 so that sink of the map machine could be synchronized to the sink of the main machine.

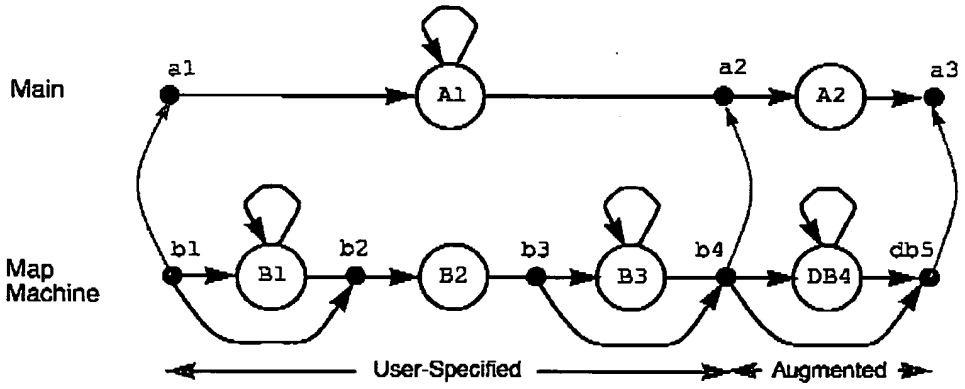


Figure 4.9: Example 1. An example to show the dot-composition process.

The result of precomposition is shown in Figure 4.10. The figure shows all the paths generated due to precomposition. As an example, let us consider the instantaneous paths out of the state vertex A1 in the main machine and state vertex B3 in the map machine. There are 6 possible combinations of instantaneous paths:

1. A1, A1 and B3, B3: Creates the instantaneous path ⟨A1, B3⟩, ⟨A1, B3⟩.
2. A1, A1 and B3, b4, DB4: Does not create a path since synchronization for b4 is not met.
3. A1, A1 and B3, b4, db5: Does not create a path since synchronization for b4 is not met.
4. A1, a2, A2 and B3, B3: Creates the path ⟨A1, B3⟩, ⟨a2, B3⟩, ⟨A2, B3⟩.
5. A1, a2, A2 and B3, b4, DB4: Creates the path ⟨A1, B3⟩, ⟨a2, B3⟩, ⟨A2, DB4⟩ since b4 is synchronized to a2.
6. A1, a2, A2 and B3, b4, db5: Does not create a path since synchronization for db5 is not met.

All state vertices are associated with action and reaction node formulas. As an example, the action and reaction node formula associated with state vertex ⟨A1, B3⟩ is the action and reaction node formula associated with state vertex B3. All event vertices are related to event vertices in the main machine. As an example, the event vertex ⟨a2, B3⟩ is related to event vertex a2 in the main machine.

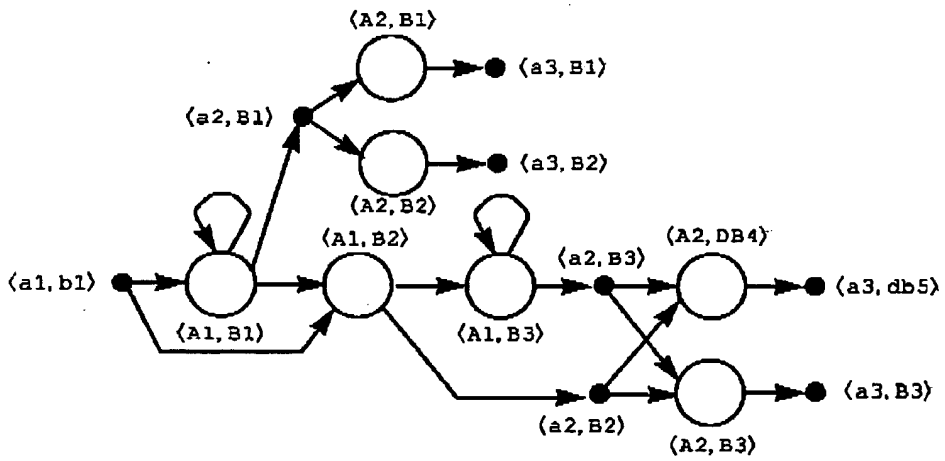


Figure 4.10: Example 1. Result of augmentation and precomposition.

4.4.3 Prune Resultant Trajectory Formula

We are interested in only those paths that start at the source vertex $\langle s_m, s_s \rangle$ and end at the sink vertex $\langle t_m, t_s \rangle$. Note that some of the paths in the graph, in Figure 4.10, do not end at the sink vertex $\langle$a3, db5$\rangle$. The last stage in the composition process prunes away all the spurious paths created by the pre-composition stage. A spurious path is defined to be a path in the graph which does not end at the sink vertex. The result of pruning is shown in Figure 4.11. This graph is the cross-product construction of the machines under restrictions specified by the synchronization function.

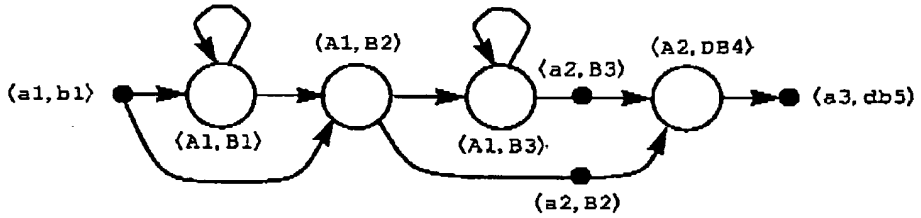


Figure 4.11: Example 1. Final result of dot-composition.

4.5 Parallel-Composition

A high-level view of the parallel-composition process is shown in Figure 4.12. We are given the trajectory formulas $\mathcal{TF}(AF_a)$ and $\mathcal{TF}(AF_b)$ [1]. Every event vertex in these trajectory formulas is related to event vertices in the main machine. The parallel-composition process will result in the trajectory formula $\mathcal{TF}(AF_a$ **and** $AF_b)$, whose event vertices will be related to the main machine.

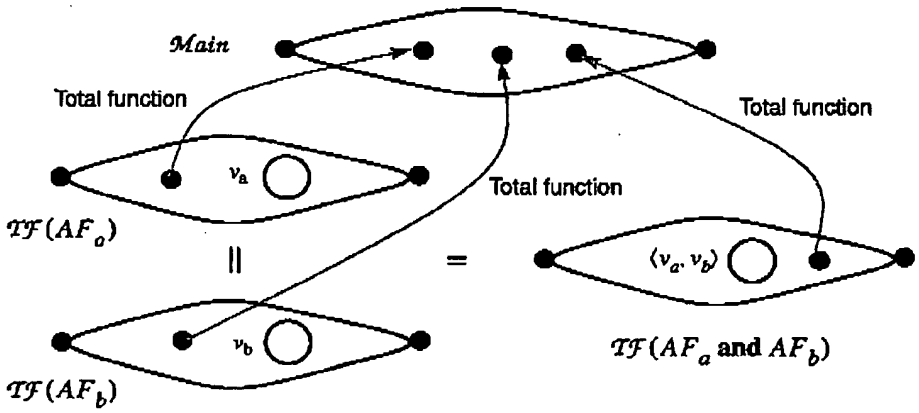


Figure 4.12: High-level view of the parallel-composition process.

1. This section uses AF_a and AF_b instead of AF_1 and AF_2 for convenience of notation.

Let $\langle V_m, U_m, E_m, s_m, t_m, \beta_m \rangle$ represent the main machine. Let $\mathcal{TF}(AF_a) = \langle V_a, U_a, E_a, s_a, t_a, \sigma_a, \sigma_r, \Upsilon_a \rangle$ represent the trajectory formula for the abstract formula AF_a and $\mathcal{TF}(AF_b) = \langle V_b, U_b, E_b, s_b, t_b, \sigma_a, \sigma_r, \Upsilon_b \rangle$ represent the trajectory formula for the abstract formula AF_b. The parallel composition will result in a trajectory formula of the form: $\langle V, U, E, s, t, \sigma_a, \sigma_r, \Upsilon \rangle$, where

- V is the set of state vertices, $V \subseteq V_a \times V_b$.
- U is the set of event vertices, $U = \{(u_a, u_b) \mid u_a \in U_a \text{ and } u_b \in U_b \text{ and } \Upsilon_a(u_a) = \Upsilon_b(u_b)\}$.
- E is the set of edges.
- s is the source, $s = \langle s_a, s_b \rangle$.
- t is the sink, $t = \langle t_a, t_b \rangle$.
- σ_a labels state vertices with action node formulas. For state vertices $v_a \in V_a$ and $v_b \in V_b$, and the state vertex $\langle v_a, v_b \rangle \in V$, the associated action node formula is $\sigma_a(\langle v_a, v_b \rangle) = \sigma_a(v_a) \text{ and } \sigma_a(v_b)$.
- σ_r labels state vertices with reaction node formulas. For state vertices $v_a \in V_a$ and $v_b \in V_b$, and the state vertex $\langle v_a, v_b \rangle \in V$, the associated reaction node formula is $\sigma_r(\langle v_a, v_b \rangle) = \sigma_r(v_a) \text{ and } \sigma_r(v_b)$.
- Υ relates event vertices to event vertices in the main machine. For vertices $u_a \in U_a$ and $u_b \in U_b$, and the event vertex $\langle u_a, u_b \rangle \in U$, the associated projection function is $\Upsilon(\langle u_a, u_b \rangle) = \Upsilon_a(u_a)$

The pseudo-code for the parallel composition algorithm is shown in Figure 4.13. The routine is called as **parallelCompose**(s_a, s_b) . Line 4 creates all the vertices. For a state vertex, the node formulas are set to $\sigma_a(\langle a_0, b_0 \rangle) = \sigma_a(a_0) \text{ and } \sigma_a(b_0)$ and $\sigma_r(\langle a_0, b_0 \rangle) = \sigma_r(a_0) \text{ and } \sigma_r(b_0)$. For an event vertex, we know that $\Upsilon_a(a_0) = \Upsilon_b(b_0)$. The event vertex is related to the main machine so that $\Upsilon(\langle a_0, b_0 \rangle) = \Upsilon_a(a_0)$. The check on line 7 assumes that the function Υ has been extended to state vertices. Each state vertex is

assumed to be a free vertex. The recursive call on line 8 is made if either a_1 and b_1 are both state vertices or both are event vertices that are related to the same event vertex in the main machine.

```
// Assumes that ∀(a ∈ V_a) , Υ_a(a) = Λ and ∀(b ∈ V_b) , Υ(b) = Λ .
vertex parallelCompose(a_0, b_0) {
    if ⟨a_0, b_0⟩ not in hash table {
        Create ⟨a_0, b_0⟩ and put in hash table.          ..... Line 4
        foreach edge (a_0, a_1) {
            foreach edge (b_0, b_1) {
                if (Υ_a(a_1) == Υ_b(b_1))                 ..... Line 7
                    ⟨a_1, b_1⟩ = parallelCompose(a_1, b_1)  ..... Line 8
            }
        }
    }
    return ⟨a_0, b_0⟩
}
```

Figure 4.13: Pseudo-code for the parallel-composition algorithm.

Consider the example shown in Figure 4.14. Event vertices in the two trajectory formulas A and B are related to event vertices in the main machine.

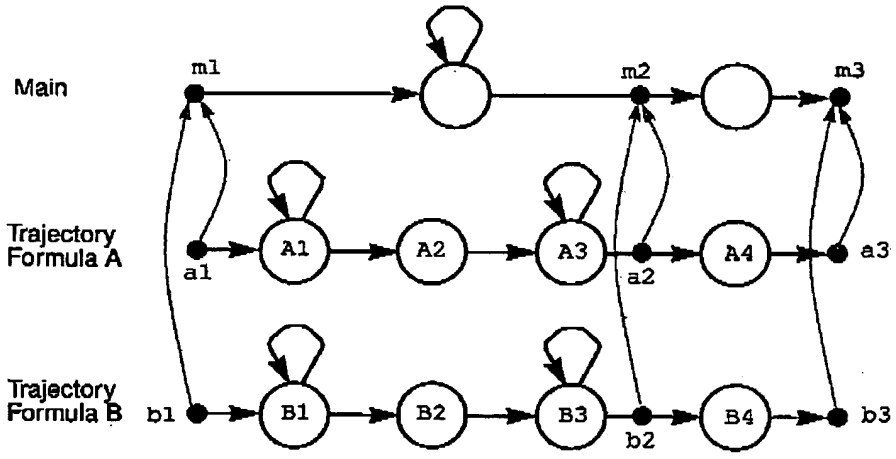


Figure 4.14: Example 2. An example to show the parallel-composition process.

The result of parallel composition of the two trajectory formulas is shown in Figure 4.15. The resultant graph is the cross-product of the two trajectory formulas under the restrictions specified by the projection function on event vertices. All event vertices are related to event vertices in the

main machine. As an example, the event vertex ⟨a2, b2⟩ is related to the event vertex m2 in the main machine. All state vertices are associated with action and reaction node formulas. As an example, the action node formula associated with state vertex ⟨A3, B2⟩ is the conjunction of the action node formulas associated with state vertices A3 and B2, and the reaction node formula is the conjunction of the reaction node formulas associated with vertices A3 and B2.

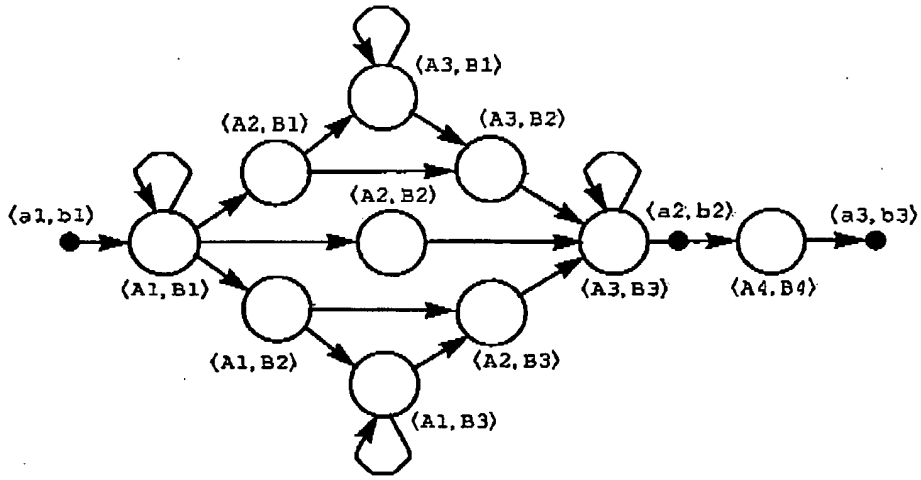


Figure 4.15: Example 2. Result of parallel-composition.

4.6 Trajectory Assertion

The trajectory assertion has essentially the same form as the trajectory formula. Assume that $\mathcal{TF}(P)$ and $\mathcal{TF}(Q)$ are the trajectory formulas associated with the precondition and postcondition respectively. The trajectory assertion is defined to be $\mathcal{TA}(P \Rightarrow Q) = \mathcal{TF}(P) // [\mathcal{TF}(Q)]^M$. A high-level view of this construction is shown in Figure 4.16. Event vertices in the trajectory formula $\mathcal{TF}(P)$ are related to event vertices in the main machine by the total function Υ_p. Event vertices in the trajectory formula $\mathcal{TF}(Q)$ are related to event vertices in the main machine by the total function Υ_q. The shift is effected by using the nextmarker function β_m in the main machine. Since β_m is a partial function, the composite function $\beta_m \cdot \Upsilon_q$ is a partial function. The two trajectory formulas are augmented so as to relate the sources and sinks of the two machines and a variation of the dot-composition operation is used to generate the trajectory assertion.

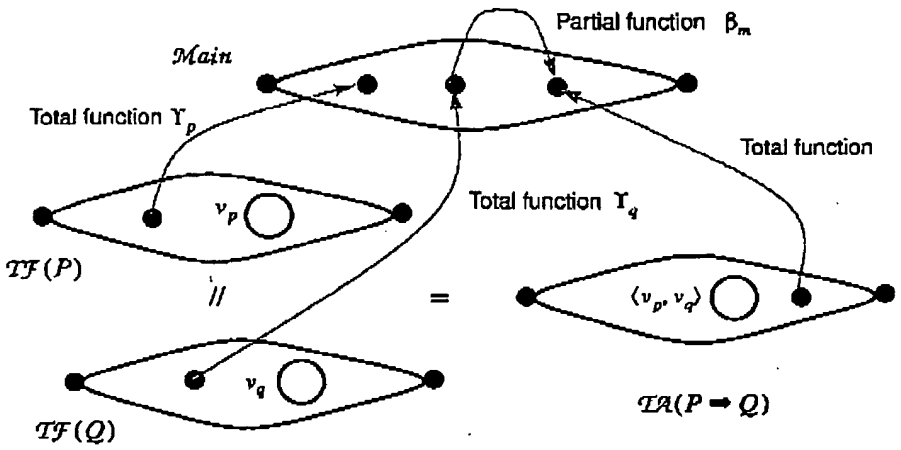


Figure 4.16: High-level view of the shift-and-compose operation.

Let $\mathcal{TF}(P) = \langle V_p, U_p, E_p, s_p, t_p, \sigma_a, \sigma_r, \Upsilon_p \rangle$ represent the *augmented* trajectory formula associated with the precondition. Let $\mathcal{TF}(Q) = \langle V_q, U_q, E_q, s_q, t_q, \sigma_a, \sigma_r, \Upsilon_q \rangle$ represent the *augmented* trajectory formula associated with the postcondition. The mirror operation reverses the role of the action and reaction node formulas, so that $[\mathcal{TF}(Q)]^M = \langle V_q, U_q, E_q, s_q, t_q, \sigma_r, \sigma_a, \Upsilon_q \rangle$. The nextmarker function in the main machine is used to shift all projections in the machine $[\mathcal{TF}(Q)]^M$. The result of the shift-and-compose operation is the trajectory assertion of the form: $\mathcal{TA}(P \Rightarrow Q) = \langle V, U, E, s, t, \sigma_a, \sigma_r, \Upsilon \rangle$, where

- V is the set of state vertices, $V \subseteq V_p \times V_q$.
- U is the set of event vertices, $U \subseteq (U_p \times U_q) \cup (U_p \times V_q) \cup (V_p \times U_q)$.
- E is the set of edges.
- s is the source, $s = \langle s_p, s_q \rangle$.
- t is the sink, $t = \langle t_p, t_q \rangle$.
- σ_a labels state vertices with action node formulas. For state vertices $v_p \in V_p$ and $v_q \in V_q$, and the state vertex $\langle v_p, v_q \rangle \in V$, the associated action node formula is $\sigma_a(\langle v_p, v_q \rangle) = \sigma_a(v_p)$ **and** $\sigma_r(v_q)$.

- σ_r labels state vertices with reaction node formulas. For state vertices $v_p \in V_p$ and $v_q \in V_q$, and the state vertex $\langle v_p, v_q \rangle \in V$, the associated reaction node formula is $\sigma_r(\langle v_p, v_q \rangle) = \sigma_r(v_p)$ and $\sigma_a(v_q)$.
- Υ relates event vertices in the trajectory assertion to event vertices in the main machine, $\Upsilon : U \rightarrow U_m$.

4.7 Example

Assume that we want to verify the bitwise-OR operation in an ALU. The abstract specification would define abstract elements, SA, SB, and ST. The state elements SA and SB serve as the source operands and ST serves as the target operand for the bitwise-OR operation. Assume that a and b are symbolic variables that denote the current values of SA and SB. The abstract assertion for the bitwise-OR operation is:

(SA is a) and (SB is b) ➡ (ST is a|b)

The specification is kept abstract. It does not specify any timing or implementation specific details. Now let's assume a specific implementation of the ALU where the source operands have to be fetched from a register file and may not be immediately available. Assume that both source operands are associated with a valid signal which specifies when the operand is available. The valid signals have a default value of logic 0 and are asserted by the register file subsystem to a logic 1 when the operand is available. The ALU computes the bitwise-OR and makes it available for storage back into the register file one cycle after both operands have been received. The block diagram for such an ALU is shown in Figure 4.17. This is a simple example that serves to illustrate some of the issues that we have encountered in our effort to verify the PowerPC architecture.

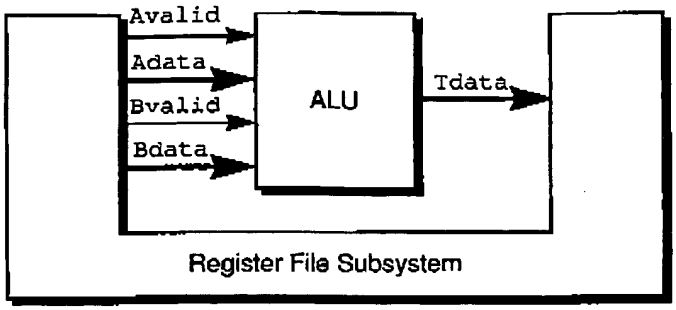


Figure 4.17: An implementation with a valid signal for source operands.

There are a few interesting points to note about this implementation: 1. The ALU might have to wait for an arbitrary number of cycles before either of the source operands is available. 2. The source operands might arrive in different orders. Figure 4.18 shows part of an execution sequence for the implementation. The sequence is divided into various segments each of which represents a bitwise-OR operation. A segment is associated with a next time point. The execution sequence can be obtained by aligning the start of the successive segment with the next time point of the current segment. Note that segments can be of different widths. Segment 1 exhibits a case where the A operand arrived before the B operand. Segments 2 and 3 exhibit cases where both operands arrived simultaneously. Segment 3 represents the maximally pipelined case where the operands were immediately available. And segment 4 exhibits a case where the B operand arrived before the A operand. The goal of verification is to ensure that the circuit correctly performs a bitwise-OR operation under any number of wait cycles and under all possible arrival orders.

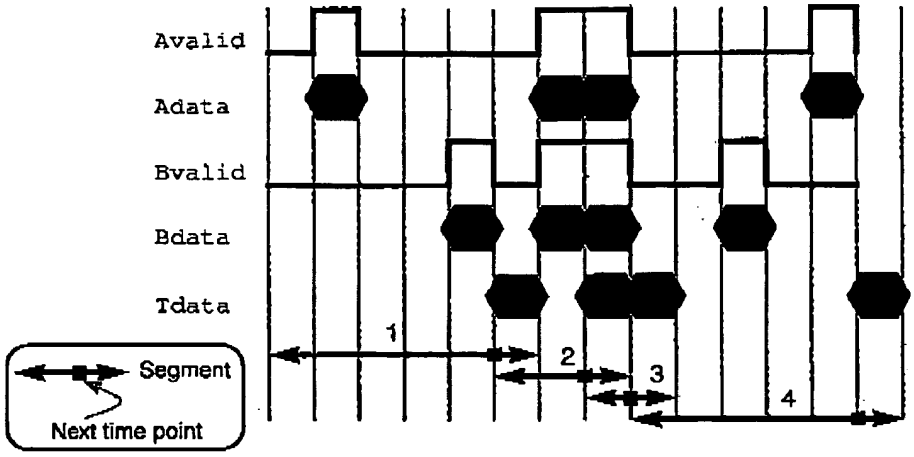


Figure 4.18: Part of an execution sequence for the bitwise-OR operation.

The main machine for such an implementation is shown in Figure 4.19. Each state vertex represents a clock cycle. One or more cycles might be required to fetch the operands. This is represented by the state vertex `fetch`. Multiple cycles are required when the operands are not immediately available. After obtaining the operands, the result of the bitwise-OR operation is available in the next cycle represented by state vertex `execute`. The main machine captures all possible segments in the execution sequence. The event vertex `fetched` represents the next time point of the segment.

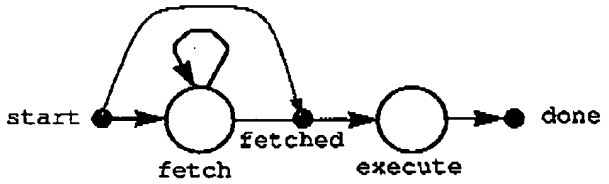


Figure 4.19: Main machine for the bitwise-OR operation.

The single-sided map machine for the abstract formula `(SA is v)` is shown in Figure 4.20. The symbolic variable `v` serves as a formal argument. Later on, the formal arguments will get replaced by the actual arguments. The logic value inside the state vertices represents the value of the valid signal for the A operand. The action node formulas associated with the state vertices are shown in the shadowed boxes in the figure. The reaction node formula is assumed to be the trivially true node formula. Since the operand might not be immediately available, the implementation might

have to wait for an arbitrary number of cycles. This is represented by state vertex `wait`. The operand is received in state vertex `fetch`. After obtaining the A operand, the implementation might have to wait again for an arbitrary number of cycles for the B operand. This is represented by state vertex `fill`. The vertices `M.start` and `M.fetched` represent the event vertices `start` and `fetched` in the main machine. So the source of the map machine is synchronized to the source of the main machine. The sink of the map machine is synchronized to the event vertex `fetched` in the main machine, where `M.fetched` represents the time point where both operands have been received.

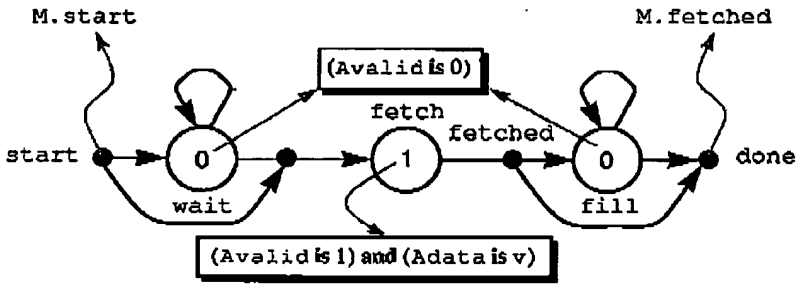


Figure 4.20: Map machine for (SA is v)

The map machine for the abstract formula (SB is v) is shown in Figure 4.21. The machine is the same as for state element SA except that the node formulas refer to the B operand. The logic value inside the state vertices represents the value of the valid signal for the B operand.

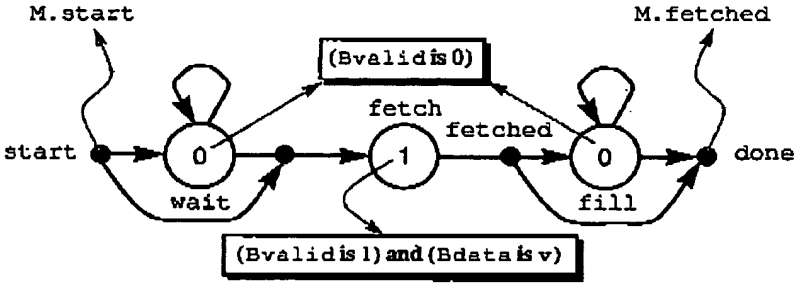


Figure 4.21: Map machine for (SB is v).

Finally, the map machine for (ST is v) is shown in Figure 4.22. The start of this machine is synchronized with the start of the main machine. This indicates that the result of the computation of the previous operation should be available at the start of the current operation.

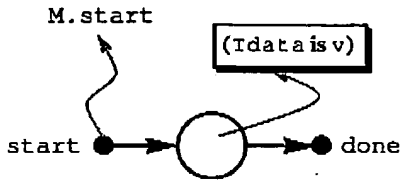


Figure 4.22: Map machine for (ST is v).

The trajectory formulas get aligned with the main machine as shown in Figure 4.23. The abstract formulas (SA is a) and (SB is b) appear in the precondition of the abstract assertion. The corresponding trajectory formulas $\mathcal{TF}$(SA is a) and $\mathcal{TF}$(SB is b) are derived from the map machines *Map*(SA is v) and *Map*(SB is v) respectively. The formal argument v in the corresponding map machines are replaced by the actual arguments a and b. The node formulas in the map machine are treated as action node formulas. Action node formulas are shown in the upper half of the state vertex. The abstract formula (ST is a|b) appears in the postcondition. The trajectory formula $[\mathcal{TF}(\text{ST is a|b})]^M$ is derived from the map machine *Map*(ST is v). The formal argument v gets replaced by the expression a|b. Due to the mirror operation, the node formula in the map machine is treated as a reaction node formula. The reaction node formula is shown in the lower half of the state vertex. The trajectory formula is shifted to the nextmarker of the source in the main machine.

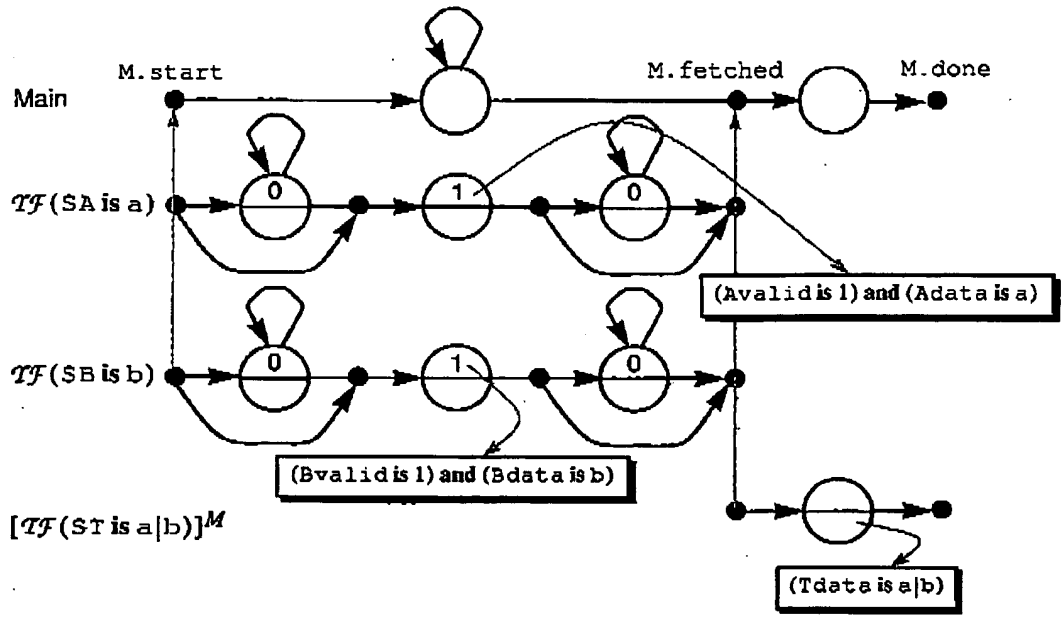


Figure 4.23: Alignment of machines for bitwise-OR operation.

The trajectory assertion for the bitwise-OR operation corresponds to the composition of these machines. For this example, composition amounts to performing the cross-product construction of the main, SA, and SB machines between event vertices `M.start` and `M.fetched`, followed by the cross-product construction of the main and ST machines between event vertices `M.fetched` and `M.done`. The implementation mapping requires one additional piece of information. Notice that the state vertices `A.fill` and `B.fill` represent the fact that one of the operands has been received and the implementation is waiting for the other. When both operands have been received, the implementation will go ahead and compute the bitwise-OR. So in the cross-product construction, we want to invalidate the cross-product of vertices `A.fill` and `B.fill`. With this additional information, the composition results in the trajectory assertion shown in Figure 4.24. The logic values in the upper half of the state vertices are the values associated with the valid signals for the A and B operand. The resultant control graph captures all possible orderings and arbitrary number of wait cycles. In the top path, the A operand was received before the B operand. In the bottom path, the B operand was received before the A operand. In the middle path, both source operands were received simultaneously. The result of the bitwise-OR operation is available in the final state vertex.

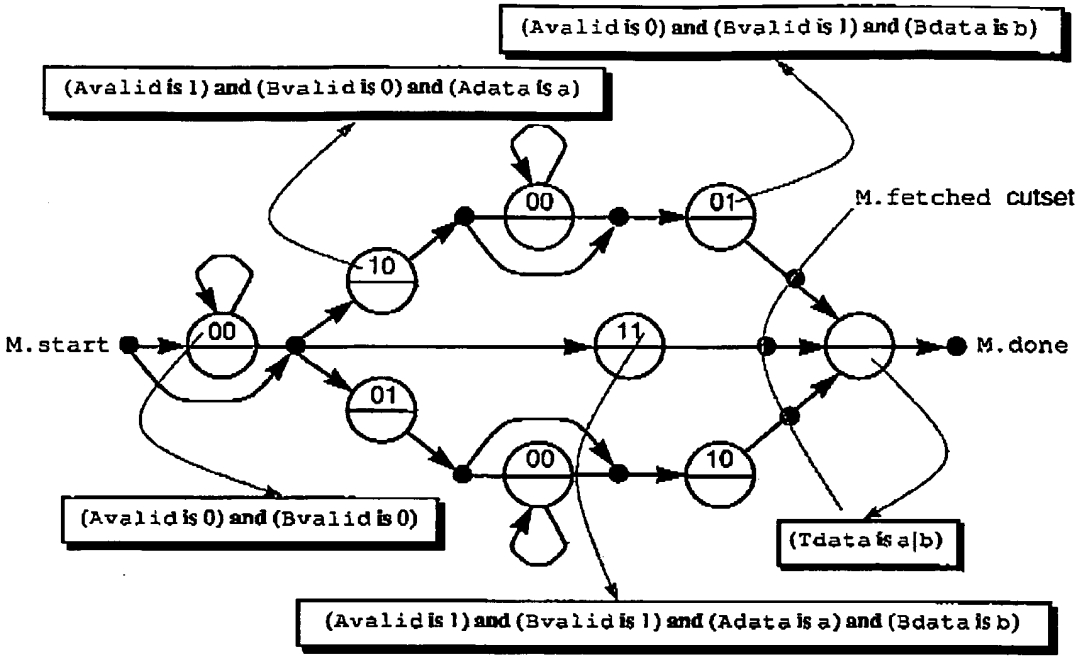


Figure 4.24: Trajectory assertion for bitwise-OR operation.

4.8 Miscellaneous Issues

In the next chapter, Symbolic Trajectory Evaluation will be used to verify the trajectory assertion on the circuit. Symbolic Trajectory Evaluation uses ordered Binary Decision Diagrams (BDDs) to perform the symbolic computation. The size of BDDs is highly dependent on the ordering of Boolean variables. Therefore, the trajectory generation needs to deal with the issue of variable ordering. Our current tool supports a simple minded user-defined variable ordering. In the future, we need to incorporate more automated and efficient techniques for variable ordering such as dynamic variable reordering[71][74].

THIS PAGE BLANK (USPTO)

Chapter 5

Verification Algorithms

The verification task is to verify that the circuit fulfills the specification. This chapter concentrates on algorithms to perform the verification task. A more rigorous understanding of what it means to verify trajectory assertions will be given later in Chapter 6.

This chapter introduces the term *Trajectory Checking* as a means to verify the trajectory assertion on a model of the circuit. The term trajectory checking has been chosen to denote its mixed heritage from model checking and Symbolic Trajectory Evaluation. The trajectory checking algorithm requires the next-state function of the entire circuit. A relaxation algorithm can be used to label the trajectory assertion with feasible circuit states. There are two forms of trajectory checking: *set-based* and *lattice-based trajectory checking*. Lattice-based trajectory checking can be viewed as an approximation of the set-based version. The lattice-based approach is computationally efficient but somewhat pessimistic. It can therefore yield false negative results, where a correct circuit fails the verification.

Trajectory checking requires the next-state function for the entire circuit. This can be very expensive for complex systems such as processors. One way of overcoming this limitation is Symbolic Trajectory Evaluation. Symbolic Trajectory Evaluation (STE) can be viewed as a modified form of symbolic simulation that computes the next-state function on-the-fly and only for that part of the circuit required by the specification[8]. STE has been used in the past to verify a limited form of the trajectory assertion. This chapter discusses extensions to STE to deal with our generalized model of trajectory assertions.

5.1 Related Work

Informally, a trajectory is sequence of states that represents an acceptable behavior of the circuit. A more rigorous definition of a trajectory will be given later in Chapter 6. Seger and Bryant intro-

duced the term trajectory assertion[18]. A simple trajectory assertion was defined in the form of an implication, i.e., antecedent implies the consequent. The antecedent and consequent were trajectory formulas that used the next-time operator to define restrictions on circuit nodes for a finite duration of time. The sequence and iteration constructs were used to create more complex trajectory assertions. In effect, their simple trajectory assertion was a single sequence of state vertices labelled with action and reaction node formulas. The iterations construct augmented the single sequence with a limited set of loops. They, however, did not allow nested or interacting loops. Our trajectory assertions are arbitrary control graphs with state vertices labelled with action and reaction node formulas.

A detailed description of related work in model checking and Symbolic Trajectory Evaluation was presented in Chapter 1.

5.2 Terminology

The mathematical form of the trajectory assertion was introduced in Section 4.6. The trajectory assertion was defined to be a control graph with action and reaction node formulas on state vertices and a function to relate event vertices to the main machine. For the purposes of this chapter, we can ignore the event vertices (except for the source and sink vertices) and their relation to the main machine. This can be done by taking a transitive closure of all the event vertices except for the source and sink. For the purposes of this chapter, a trajectory assertion can be assumed to be a tuple of the form: $G = \langle V, U, E, s, t, \sigma_a, \sigma_r \rangle$, where

- $\langle V, U, E, s, t \rangle$ is a control graph, where $U = \{s, t\}$.
- σ_a labels state vertices with action node formulas.
- σ_r labels state vertices with reaction node formulas.

In this chapter, we shall refer to the set W as the set of all vertices, $W = V \cup \{s\} \cup \{t\}$.

5.3 Set-Based Trajectory Checking

Trajectory checking uses a relaxation algorithm to label the trajectory assertion with feasible circuit states. The set-based approach operates on a set-based model structure of the circuit and a set-based interpretation of the trajectory assertions.

5.3.1 Set-Based Model Structure

Assume that N_v is the set of node elements in the circuit. The set of node elements is the set of external inputs, outputs, and internal nets in the circuit. The set of node assignments is defined to be $N = \{0, 1\}^m$, where $m = |N_v|$. The circuit is modeled as a tuple: $\langle N, \eta \rangle$, where η is the *circuit excitation function*, $\eta : N \rightarrow 2^N$. The excitation function is defined in a non-traditional way. Given the current node assignment, the excitation function expresses constraints on the set of next node assignments. Since the value of an input is controlled by the external environment, the circuit itself does not impose any constraint on the inputs. The circuit, however, does impose constraints on the outputs and internal state elements. The excitation function expresses the constraints on the set of node assignments after the circuit has reached a stable state assuming all external inputs are held fixed until the circuit has reached that stable state.

As an example, consider an inverter realization. The set of node assignments is $N = \{00, 01, 10, 11\}$, where the left bit represents the logic value on the input node and the right bit represents the value on the output node of the inverter. The excitation function for the inverter is shown in Figure 5.1. Consider the case of the input currently at logic 0 and output at logic 1. That is represented by the node assignment 01 in the figure. At the next time instant, the output is constrained to be at logic 1, whereas the input is unconstrained. That is represented by transitions from node assignment 01 to node assignments 01 and 11.

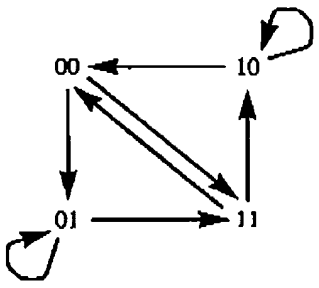


Figure 5.1: Set-based excitation function for an inverter.

The set-based model structure for a circuit is a tuple of the form: $C = \langle N, \delta \rangle$, where N is the set of node assignments and δ is the *model excitation function*, $\delta : 2^N \rightarrow 2^N$. The circuit excitation function is extended to form the model excitation function in the following way: $\delta(A) = \bigcup_{a \in A} \eta(a)$, where $A \subseteq N$.

5.3.2 Set-Based Trajectory Assertion

In the set-based model, a node formula is represented by a set of node assignments. The set of node assignments corresponding to a node formula F, written $Set(F)$, is defined recursively,

- $Set(true) = N$
- $Set(n_i \text{ is } 0)$ is the subset of N containing 2^{m-1} node assignments having the i^{th} position as 0. Similarly $Set(n_i \text{ is } 1)$ is the subset of N containing 2^{m-1} node assignments having the i^{th} position as 1.
- $Set(F_1 \text{ and } F_2) = Set(F_1) \cap Set(F_2)$
- $Set(0 \rightarrow F) = N$ and $Set(1 \rightarrow F) = Set(F)$

For a vertex $v \in V$, we will refer to the set $Set(\sigma_a(v))$ as the set of action node assignments and the set $Set(\sigma_r(v))$ as the set of reaction node assignments associated with the vertex v.

We can classify trajectory assertions into three categories in increasing order of expressiveness and complexity: 1) *Oblivious* 2) *Adaptive* and 3) *Prescient*. Consider a vertex v in the trajectory assertion and pick any two vertices v_i and v_j such that $(v, v_i) \in E$ and $(v, v_j) \in E$. An oblivious trajectory assertion has the restriction that $Set(\sigma_a(v_i)) \cap Set(\sigma_a(v_j)) = \emptyset$. In other words, the

next stimulus or constraint imposed by the environment defines a unique vertex in the trajectory assertion. An adaptive trajectory assertion has the restriction that either $Set(\sigma_a(v_i)) \cap Set(\sigma_a(v_j)) = \emptyset$ or $Set(\sigma_r(v_i)) \cap Set(\sigma_r(v_j)) = \emptyset$. The next stimulus and the next circuit response together define a unique vertex in the trajectory assertion. This can be viewed as a lookahead-1 trajectory assertion since it requires knowledge of the next circuit response. A prescient trajectory assertion does not place any such restrictions. Such a trajectory assertion requires knowledge of the circuit response in the future to determine the next vertex in the graph.

In this chapter, we limit ourselves to algorithms to verify oblivious trajectory assertions. A more rigorous definition of what it means to verify prescient trajectory assertions will be given later in Chapter 6. Possible extensions to the algorithm to deal with adaptive trajectory assertions are presented as future work in Chapter 9.

The next section defines a relaxation algorithm that can be used to verify oblivious assertions without explicitly enumerating all paths in the graph.

5.3.3 Set-Based Relaxation Algorithm

The relaxation algorithm uses the model structure to compute the *defining trajectory* for an oblivious trajectory assertion. Informally, a trajectory is sequence of states that represents an acceptable behavior of the circuit. A more rigorous definition of a trajectory will be given later in Chapter 6. The defining trajectory is the sequence of maximal set of states that satisfy the action node formulas. The verification task is to compute the defining trajectory and verify that each of the defining trajectory satisfies the reaction node formulas. The relaxation algorithms computes the defining trajectory in terms of *defining trajectory labels* on each vertex in the trajectory assertion. The defining trajectory labels map each vertex in the trajectory assertion to a set of node assignments, $\kappa : W \rightarrow 2^N$.

The pseudo-code for the set-based relaxation algorithm is shown in Figure 5.2. Line 4 initializes the defining trajectory labels to be the empty set for all vertices. Line 5 initializes the defining trajectory label for the source vertex to be the set of all node assignments. The relaxation algorithm

starts from the source vertex and works its way to the sink vertex in a depth-first manner. Line 10 computes the effect of vertex w on the vertex w_i, where (w, w_i) is an edge in the control graph. The temporary variable Y stores the set of node assignments resulting from applying the excitation function on the defining trajectory label on vertex w and intersecting that with the set of action node assignments on vertex w_i. Lines 11 and 12, in effect, compute the least fixed point of the defining trajectory label on vertex w_i. The least fixed point computation corresponds to performing a reachability analysis on the set of node assignments. For the verification to succeed, the defining trajectory label on a vertex should be contained in the reaction node assignments for that vertex.

```
// G = ⟨V, U, E, s, t, σ_a, σ_r⟩
// W = V ∪ {s} ∪ {t}
verify(G) {
   foreach (w ∈ W) κ(w) = ∅ ..... Line 4
   κ(s) = N ..... Line 5
   relax(G, s)
}
relax(G, w) {
   foreach ( (w, w_i) ∈ E ) {
      Y = δ(κ(w)) ∩ Set(σ_a(w_i)) ..... Line 10
      if ( Y ⊈ κ(w_i) ) { ..... Line 11
         κ(w_i) = κ(w_i) ∪ Y ..... Line 12
         if ( κ(w_i) ⊆ Set(σ_r(w_i)) ) relax(G, w_i)
         else "verification failed"
      }
   }
}
```

Figure 5.2: Set-based relaxation algorithm for oblivious trajectory assertions.

As an example, consider a circuit for a modulo-3 counter. The state diagram for a modulo-3 counter is shown in Figure 5.3. The state diagram has 4 states A, B, C and D which are encoded in the circuit as 00, 01, 10, and 11 respectively. Assume that A is the start state. The circuit has an active high `reset` signal. The transitions in the state diagram are labelled with the `reset` signal value. When the `reset` signal is at logic 1, the circuit resets into the start state A. When the `reset` signal is at logic 0, the counter performs a modulo-3 count, i.e., transitions through the

states A, B, C, A ... Apart from the external input reset signal and two internal state bits, let us assume that the circuit has another external input in signal and an output out signal. Assume that the output is the AND of the two state bits and the input in signal. Thus when input in is 1, out will be 0 for states A, B, C and 1 for state D.

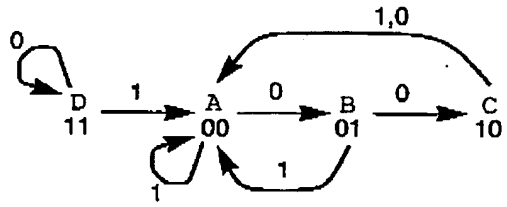


Figure 5.3: State diagram for a modulo-3 counter.

Let us assume that we are trying to verify the trajectory assertion shown in Figure 5.4. The action node formulas are shown in the upper half and the reaction node formulas are shown in the lower half of the shadowed box. The trajectory assertion is stating the following property: If the circuit is reset for one cycle, followed by maintaining the reset signal to logic 0 for an arbitrary number of cycles, then in the next cycle when in goes to 1, the output out should be logic 0, i.e., the circuit should be in states A, B, or C.

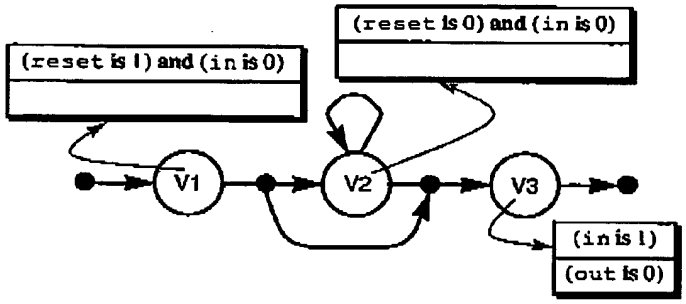


Figure 5.4: Trajectory assertion for modulo-3 counter.

There are 5 node elements in the circuit, namely the reset signal, in signal, the 2 internal state bits and out signal. Therefore, the set of node assignments is $N = \{0,1\}^5 = \{00000, \ldots, 11111\}$, where the leftmost bit is the logic value associated with the reset signal, the next bit is the logic value associated with the in signal, the next 2 bits are the logic values associated with the internal state bits and the rightmost bit is the logic value associated with the out signal. The set-based circuit excitation function can be easily derived from the

state diagram in Figure 5.3. As an example, if the current `reset` is 0, `in` is 0, and the current state is A(00), then the next inputs are unconstrained, the next state is constrained to be B(01) and next output is constrained to be 0, i.e. $\eta(00000) = \{00010, 01010, 10010, 11010\}$. The circuit excitation function will be extended to form the model excitation function δ. The node formulas in the trajectory assertion in Figure 5.4 are interpreted as a set of node assignments. As an example, the set of node assignments associated with the action node formula on state vertex V2 is the set of node assignments where the reset and input signals are at logic 0, i.e., $Set(\sigma_a(V2)) = \{00000, 00001, 00010, 00011, 00100, 00101, 00110, 00111\}$.

The relaxation algorithm will compute the defining trajectory labels for state vertices in the trajectory assertion. Let us first consider the vertex V1. The defining trajectory label for the vertex V1 is computed as follows:

$$\begin{aligned}\kappa(V1) &= \delta(N) \cap Set(\sigma_a(V1)) \\ &= \{10000, 10010, 10100, 10110\}\end{aligned}$$

The defining trajectory label for the vertex V2 is computed as follows:

$$\begin{aligned}\kappa^1(V2) &= \delta(\kappa(V1)) \cap Set(\sigma_a(V2)) \\ &= \{00000\} \\ \kappa^2(V2) &= \kappa^1(V2) \cup [\delta(\kappa^1(V2)) \cap Set(\sigma_a(V2))] \\ &= \{00000\} \cup \{00010\} = \{00000, 00010\} \\ \kappa^3(V2) &= \kappa^2(V2) \cup [\delta(\kappa^2(V2)) \cap Set(\sigma_a(V2))] \\ &= \{00000, 00010\} \cup \{00100\} \\ &= \{00000, 00010, 00100\}\end{aligned}$$

The set $\kappa^3(V2)$ represents the least fixed point since $\kappa^3(V2) = \kappa^4(V2)$. Therefore, the defining trajectory label for vertex V2 is $\kappa(V2) = \kappa^3(V2)$. The least fixed point calculation on the self loop on state vertex V2 amounts to performing a reachability analysis, i.e. the circuit states that are reachable in vertex V2. Thus the reachable states in the circuit are A(00), B(01) and C(10).

The defining trajectory label for the vertex V3 is computed as follows:

$$\begin{aligned}\kappa^1(V3) &= \delta(\kappa(V1)) \cap Set(\sigma_a(V3)) \\ &= \{01000, 11000\} \\ \kappa^2(V3) &= \kappa^1(V3) \cup (\delta(\kappa(V2)) \cap Set(\sigma_a(V3))) \\ &= \{01000, 11000\} \cup \{01000, 01010, 01100, 11000, 11010, 11100\} \\ &= \{01000, 01010, 01100, 11000, 11010, 11100\}\end{aligned}$$

Therefore, the defining trajectory label for vertex V3 is $\kappa(V3) = \kappa^2(V3)$. The trajectory assertion will be reported to be true since the output signal is 0 in all the node assignments in $\kappa(V3)$. In other words $\kappa(V3) \sqsubseteq Set(\sigma_r(V3))$.

5.4 Lattice-Based Trajectory Checking

Lattice-based trajectory checking can be viewed as an approximation of the set-based approach. The logic set is extended to the ternary domain, $\{0, 1, X\}$. The logic X denotes an unknown and possibly indeterminate logic value. The logic values are assumed to have a partial order with logic X lower in the partial order than both logic 0 and logic 1 as shown in Figure 5.5. The partial order operator $\sqsubseteq$ is defined as follows: $a \sqsubseteq a$ for any ternary value a, $X \sqsubseteq 0$ and $X \sqsubseteq 1$.

Figure 5.5: Partial order for logic 0, 1, and X.

We say that two ternary values a and b are compatible, denoted $a \sim b$, when there is some ternary value c such that $a \sqsubseteq c$ and $b \sqsubseteq c$. In other words, two values are compatible unless one is 0 and the other is 1.

We can define two operations, the least upper bound and greatest lower bound, over ternary values. The symbols $\sqcup$ and $\sqcap$ denote the least upper bound and greatest lower bound operations respectively. Assuming two ternary values a and b, the least upper bound and greatest lower bound operations are defined as follows:

$a \sqcup b$	0	1	X
0	0	-	0
1	-	1	1
X	0	1	X

$a \sqcap b$	0	1	X
0	0	X	X
1	X	1	X
X	X	X	X

Note that the least upper bound is defined only under the case that a and b are compatible. The - in the table for $a \sqcup b$ denotes incompatible cases where the least upper bound is not defined.

5.4.1 Lattice-Based Model Structure

The set of node assignments is extended to $\{0, 1, X\}^m$, where $m = |N_v|$. The elements of the set define a lattice. A top element $\top$ is introduced to complete the lattice. Therefore, the set of node assignments is $\tilde{N} = \{0, 1, X\}^m \cup \top$. The elements of the set $\tilde{N}$ define a lattice $[\tilde{N}, \sqsubseteq]$, with X^m as the bottom element and $\top$ as the top element. The partial order in the lattice is a pointwise extension of the partial order shown in Figure 5.5. The compatible, least upper bound, and greatest lower bound operators are also extended pointwise. The least upper bound of two lattice elements $\tilde{A}$ and $\tilde{B}$ is defined to be the top element when $\tilde{A}$ and $\tilde{B}$ are *not* compatible with each other. The complete lattice for $m = 2$ is shown in Figure 5.6. As an example, $0X \sqsubseteq 01$, which should interpreted as the lattice element $0X$ has less information than the lattice element 01. Also $0X \sqcup X1 = 01$, $0X \sqcup 1X = \top$ and $00 \sqcap 01 = 0X$.

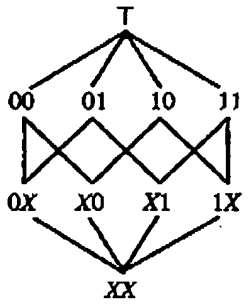


Figure 5.6: Complete lattice for 2 node elements.

The lattice-based formulation can be viewed as an approximation of the set-based formulation. The relation between symbols and operators in the two formulations is shown in the table below. A

set is approximated by an element of the lattice. As an example, the set $\{00, 01\}$ would be accurately represented as $00 \sqcap 01 = 0X$. However, the set $\{01, 10\}$ would have to be approximated to $01 \sqcap 10 = XX$.

Set	Lattice
$A \subseteq N$	$\tilde{A} \in \bar{N}$
N	X^m
$\varnothing$	$\top$
$\subseteq$	$\sqsupseteq$
$\cap$	$\sqcup$
$\cup$	$\sqcap$

The lattice-based model structure for a circuit is a tuple of the form: $\tilde{C} = \langle [\bar{N}, \sqsubseteq], \bar{\delta} \rangle$, where $\bar{\delta}$ is the model excitation function, $\bar{\delta} : \bar{N} \rightarrow \bar{N}$ and $\bar{\delta}(\top) = \top$. The model excitation function is required to be monotone over the partial order in the lattice, i.e if $\tilde{A} \sqsubseteq \tilde{B}$, then $\bar{\delta}(\tilde{A}) \sqsubseteq \bar{\delta}(\tilde{B})$. The monotonicity requirement is consistent with our use of information content. If a function is monotone, we cannot gain any information by reducing the information content of the arguments to the function. In other words, changing some signal from binary values to X will either have no effect on the excitation, or it will change some binary value to X.

Again, consider an inverter realization. The set of node assignments and the associated lattice is shown in Figure 5.6, where the left bit represents the logic value on the input node and the right bit represents the logic value on the output node of the inverter. The lattice based excitation function for the inverter is shown in Figure 5.7. Consider the case of the input currently at logic 0 and output is at logic 1. That is represented by lattice element 01 in the figure. At the next time instant, the output is constrained to be at logic 1, where as the input is unconstrained. That is represented by a transition from lattice element 01 to lattice element $X1$.

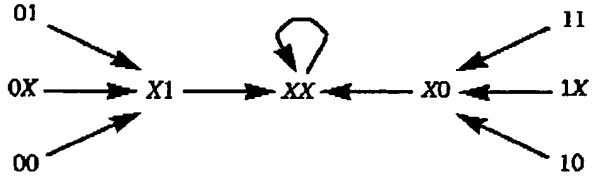


Figure 5.7: Lattice-based excitation function for an inverter.

5.4.2 Lattice-Based Trajectory Assertion

In the lattice-based model, a node formula is represented by an element of the lattice. The restricted form of the node formula ensures that it can be represented by a unique lattice element. The lattice element corresponding to a node formula F, written $\mathcal{L}at(F)$, is defined recursively:

- $\mathcal{L}at(true) = X^m$.
- $\mathcal{L}at(n_i \text{ is } 0)$ is an element of the lattice with the i^{th} position in the element being 0 and the rest of the positions being X's. $\mathcal{L}at(n_i \text{ is } 1)$ is an element of the lattice with the i^{th} position being 1 and the rest of the positions being X's.
- $\mathcal{L}at(F_1 \text{ and } F_2) = \mathcal{L}at(F_1) \sqcup \mathcal{L}at(F_2)$.
- $\mathcal{L}at(0 \rightarrow F) = X^m$ and $\mathcal{L}at(1 \rightarrow F) = \mathcal{L}at(F)$.

For a vertex $v \in V$, we will refer to the element $\mathcal{L}at(\sigma_a(v))$ as the action lattice element and the element $\mathcal{L}at(\sigma_r(v))$ as the reaction lattice element associated with the vertex v. Observe that the above construction is exact. The approximation in the lattice-based method is due to the greatest lower bound operation that will be used in the relaxation algorithm.

Again, we will restrict ourselves to oblivious trajectory assertions. Consider a vertex v in the trajectory assertion and pick any two vertices v_i and v_j such that $(v, v_i) \in E$ and $(v, v_j) \in E$. In the lattice domain, the restriction for an oblivious trajectory assertion translates into $\mathcal{L}at(\sigma_a(v_i)) \sqcup \mathcal{L}at(\sigma_a(v_j)) = \top$. The next section defines a lattice-based relaxation algorithm that can be used to verify oblivious assertions without explicitly enumerating all paths in the graph.

5.4.3 Lattice-Based Relaxation Algorithm

The relaxation algorithm computes the defining trajectory in terms of *defining trajectory labels* on each vertex in the oblivious trajectory assertion. The defining trajectory is the unique weakest trajectory that satisfies the action node formulas. The defining trajectory labels map each vertex in the trajectory assertion to a lattice element, $\tilde{\kappa} : W \rightarrow \bar{N}$.

The pseudo-code for the lattice-based relaxation algorithm is shown in Figure 5.8. Line 4 initializes the defining trajectory labels to be the top lattice element for all vertices. Line 5 initializes the defining trajectory label for the source vertex to be the bottom lattice element. The relaxation algorithm starts from the source vertex and works its way to the sink vertex in a depth-first manner. Line 10 computes the effect of vertex w on the vertex w_i, where (w, w_i) is an edge in the control graph. The temporary variable $\tilde{Y}$ stores the lattice element resulting from applying the excitation function on the defining trajectory label on vertex w and taking the least upper bound with the action lattice element on vertex w_i. Lines 11 and 12, in effect, compute the greatest fixed point of the defining trajectory label on vertex w_i. The greatest fixed point computation corresponds to performing an approximate reachability analysis. For the verification to succeed, the defining trajectory label on a vertex should be the same or higher in the partial order than the reaction lattice element for that vertex.

```
// G = ⟨V, U, E, s, t, σ_a, σ_r⟩
// W = V ∪ U
verify(G) {
    foreach (w ∈ W) κ̃(w) = ⊤                    ..... Line 4
    κ̃(s) = X^m                                  ..... Line 5
    relax(G, s)
}
relax(G, w) {
    foreach ( (w, w_i) ∈ E ) {
        Ỹ = δ̃(κ̃(w)) ⊔ Lat(σ_a(w_i))            ..... Line 10
        if ( Ỹ ⋢ κ̃(w_i) ) {                      ..... Line 11
            κ̃(w_i) = κ̃(w_i) ⊓ Ỹ               ..... Line 12
            if ( κ̃(w_i) ⊒ Lat(σ_r(w_i)) ) relax(G, w_i)
            else "verification failed"
        }
    }
}
```

Figure 5.8: Lattice-based relaxation algorithm for oblivious trajectory assertions.

The lattice-based approach is computationally more efficient than the set-based approach. The reason is that sets can now be represented by a single lattice element. The logic value X is used to model nondeterminism in the system. The lattice based approach, however, can lead to a pessimis-

tic verification. A pessimistic verification can generate a false negative, i.e., a correct circuit can be reported to be incorrect.

Consider the example of the modulo-3 counter that was introduced in Section 5.3.3. The set of node assignments is $\tilde{N} = \{0, 1, X\}^5 \cup \top$. The elements of this set define a complete lattice. The model excitation function can be derived from the state diagram in Figure 5.3. As an example, if the current `reset` is 0, `in` is 0, and the current state is A(00), then the next input is unconstrained, the next state is constrained to be B(01), and the next output is constrained to be 0, i.e., $\tilde{\delta}(00000) = XX010$. The node formulas in the trajectory assertion in Figure 5.4 will be interpreted as lattice elements. As an example, the lattice element associated with the action node formula on state vertex V2 is $\mathcal{L}at(\sigma_a(V2)) = 00XXX$.

The relaxation algorithm will compute the defining trajectory labels for state vertices in the trajectory assertion. Let us first consider the vertex V1. The defining trajectory label for vertex V1 is computed as follows:

$$\begin{aligned}\tilde{\kappa}(V1) &= \tilde{\delta}(X^m) \sqcup \mathcal{L}at(\sigma_a(V1)) \\ &= 10XX0\end{aligned}$$

The defining trajectory label for the vertex V2 is computed as follows:

$$\begin{aligned}\tilde{\kappa}^1(V2) &= \tilde{\delta}(\tilde{\kappa}(V1)) \sqcup \mathcal{L}at(\sigma_a(V2)) \\ &= XX000 \sqcup 00XXX = 00000 \\ \tilde{\kappa}^2(V2) &= \tilde{\kappa}^1(V2) \sqcap [\tilde{\delta}(\tilde{\kappa}^1(V2)) \sqcup \mathcal{L}at(\sigma_a(V2))] \\ &= 00000 \sqcap (XX010 \sqcup 00XXX) = 000X0 \\ \tilde{\kappa}^3(V2) &= \tilde{\kappa}^2(V2) \sqcap [\tilde{\delta}(\tilde{\kappa}^2(V2)) \sqcup \mathcal{L}at(\sigma_a(V2))] \\ &= 000X0 \sqcap (XXXXX \sqcup 00XXX) = 00XXX\end{aligned}$$

The lattice element $\tilde{\kappa}^3(V2)$ represents the greatest fixed point since $\tilde{\kappa}^3(V2) = \tilde{\kappa}^4(V2)$. Therefore, the defining trajectory label for vertex V2 is $\tilde{\kappa}(V2) = \tilde{\kappa}^3(V2)$. Note that the lattice-based approach has erred on the side of pessimism and has approximated the reachable set of states by the lattice element *XX*. The lattice element *XX* corresponds to the states A, B, C, and D. The lattice based approach calculates the reachable lattice element to be *XX* even though the set-based approach has revealed that the reachable states are only A, B, and C.

The defining trajectory label for vertex V3 is computed as follows:

$$\begin{aligned}\tilde{\kappa}^1(V3) &= \tilde{\delta}(\tilde{\kappa}(V1)) \sqcup \mathcal{L}at(\sigma_a(V3)) \\ &= XX000 \sqcup X1XXX = X1000 \\ \tilde{\kappa}^2(V3) &= \tilde{\kappa}^1(V3) \sqcap [\tilde{\delta}(\tilde{\kappa}(V2)) \sqcup \mathcal{L}at(\sigma_a(V3))] \\ &= X1000 \sqcap (XXXXX \sqcup X1XXX) = X1XXX\end{aligned}$$

The trajectory assertion will be reported to be false since $\tilde{\kappa}(V3)$ is lower in the partial order than the reaction lattice element on state vertex V3. In other words $\tilde{\kappa}(V3) \not\sqsupseteq \mathcal{L}at(\sigma_r(V3))$. In other words the lattice-based approach reports a correct circuit to be incorrect.

The above example has revealed a shortcoming of the lattice-based approach. The lattice-based approach errs on the side of pessimism for the sake of efficiency. A reason for this pessimism is the greatest lower bound operator. The greatest lower bound operator $\sqcap$ on lattice elements approximates the union operator $\cup$ on sets. In our example, the reachable set of states $\{00, 01, 10\}$ had to be approximated by the lattice element $00 \sqcap 01 \sqcap 10 = XX$. The greatest lower bound operation is used to obtain a monotone excitation function. Also, the greatest lower bound is used on line 12 of the relaxation algorithm in Figure 5.8.

The set and lattice-based approaches can be viewed as two ends of a spectrum. At one end of the spectrum is the set-based approach that leads to an accurate verification but is computationally complex. At the other end of the spectrum is the lattice-based approach that is computationally efficient but can lead to a pessimistic verification. The reason for efficiency is that a set can be represented by a single lattice element. The efficiency is obtained at the price of a pessimistic verification. A pessimistic verification can generate a false negative, i.e., a correct circuit is reported to be incorrect. A pessimistic verification, however, will never generate a false positive, i.e., the verification will never report an incorrect circuit to be correct.

5.5 Symbolic Trajectory Evaluation

Trajectory checking requires the model excitation function for the entire circuit. This can be very expensive. As an example, it would be impossible to obtain the complete excitation function for complex processors. One way of overcoming this limitation is to use a simulation-based verifica-

tion strategy. A simulation-based verifier computes the excitation function on-the-fly and only for that part of the circuit required by the trajectory assertion. Symbolic Trajectory Evaluation (STE) is an extension of symbolic simulation that provides a concrete way of verifying the desired system behavior over a period of time[8][12][18]. A description of the ternary symbolic representation used in STE is given in Section 5.5.1.

Oblivious trajectory assertions can be classified into three categories in increasing order of generalization and complexity: 1) *Single Sequence* 2) *Acyclic* and 3) *Generalized.* The control graph associated with a single sequence trajectory assertion has a linear sequence of vertices. An acyclic trajectory assertion has a directed acyclic control graph. And the generalized trajectory assertion has any arbitrary control graph. In earlier work, STE has been used to verify single sequence trajectory assertions[8][12]. A brief description of earlier work is given in Section 5.5.2. We have extended STE to verify generalized trajectory assertions. Our extensions for acyclic and generalized trajectory assertions are described in Section 5.5.3 and Section 5.5.4 respectively.

5.5.1 Ternary Symbolic Representation

STE represents a ternary value, $y \in \{0, 1, X\}$, with a "dual rail" encoding, (y^h, y^l), where $y^h, y^l \in \{0, 1\}$. The symbols y^h and y^l refer to the high and low rails of the encoding. The encoding for the ternary value is given below. The encoding (0, 0) serves as a don't care.

y	y^h	y^l
0	0	1
1	1	0
X	1	1
D	0	0

The encoding can be extended into the symbolic domain. In the symbolic domain, Boolean encodings are defined as a function over a set of symbolic variables. The Boolean encodings (y^h, y^l) denote a ternary symbolic function y. The partial order, compatible, least upper bound, and greatest upper bound operations are extended into the symbolic domain. Let a and b represent two ternary symbolic functions with Boolean encodings (a^h, a^l) and (b^h, b^l) respectively. The partial

order $a \sqsubseteq b$ is defined as the Boolean function $a^h \cdot b^h + a^h \cdot \overline{b^l} + a^l \cdot \overline{b^h}$. Compatibility of two ternary symbolic functions $a \sim b$ is defined as the Boolean function $a^h \cdot b^h + a^l \cdot b^l$.

The least upper bound operator $a \sqcup b$ is defined in terms of the Boolean encodings as $(a^h \cdot b^h, a^l \cdot b^l)$. A Boolean *OK* function defines the case where the two ternary symbolic functions, a and b, are compatible with each other. The greatest lower bound operator $a \sqcap b$ is defined in terms of the Boolean encodings as $(a^h + b^h, a^l + b^l)$.

As an example, consider $\mathtt{a} = (\mathtt{u}, \bar{\mathtt{u}})$ and $\mathtt{b} = (\mathtt{v}, \bar{\mathtt{v}})$, where u and v are Boolean symbolic variables. The least upper bound is $\mathtt{a} \sqcup \mathtt{b} = (\mathtt{u} \oplus \mathtt{v}, \mathtt{u} \cdot \mathtt{v}, \bar{\mathtt{u}} \cdot \bar{\mathtt{v}})$. The table below shows the cases due to all possible assignments to the Boolean variables u and v. The shaded rows represent the case where a and b are not compatible with each other.

u	v	a ⊔ b	a ~ b
0	0	0	1
0	1	*D*	0
1	0	*D*	0
1	1	1	1

We can define a ternary existential quantification operator $\overset{*}{\underset{V}{\exists}} a$ for the ternary symbolic function a over the set of symbolic variables V. The Boolean encodings for the ternary existential quantification operator is defined to be the existential quantification of the Boolean encoding, i.e., $\left(\overset{*}{\underset{V}{\exists}} a\right)^h = \underset{V}{\exists}(a^h)$ and $\left(\overset{*}{\underset{V}{\exists}} a\right)^l = \underset{V}{\exists}(a^l)$.

As an example, consider that $\mathtt{a} = (\mathtt{u}, \bar{\mathtt{u}} + \bar{\mathtt{v}})$. Then $\overset{*}{\underset{\mathtt{v}}{\exists}} \mathtt{a} = (\mathtt{u}, 1)$, i.e., the ternary existential quantification is restricted to the values 0 and *X*. The table below shows the cases due to all possible assignments to the Boolean variables u and v.

u	v	a	$\overset{*}{\underset{\mathtt{v}}{\exists}}$a
0	0	0	0
0	1	0	0
1	0	*X*	*X*
1	1	1	*X*

A ternary symbolic function is extended pointwise to represent a symbolic element in a lattice. All the operators are extended pointwise to operate on these symbolic lattice elements.

5.5.2 Single Sequence Trajectory Assertion

The general form of a single sequence trajectory assertion is shown in Figure 5.9. The trajectory assertion has a linear sequence of state vertices v_1 to v_k.

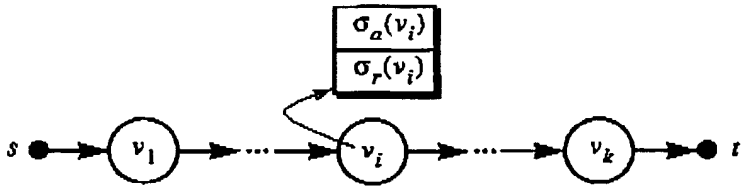


Figure 5.9: General form of a single sequence trajectory assertion.

STE uses a variation of the trajectory checking algorithm shown in Figure 5.8 to verify single sequence trajectory assertions. All the lattice operations are extended into the symbolic domain. Instead of precomputing the entire model excitation function $\tilde{\delta}$, the simulator is used to obtain the next-state of the circuit at each step of the iteration. The defining trajectory labels are stored as symbolic values on nets in the simulator. The essence of the STE algorithm is captured in the algorithm shown in Figure 5.10. The global variable $\tilde{Q}$ represents net values in the simulator. All net values are initialized to the logic value *X*. STE maintains two global Boolean functions OK_A and OK_R defined over symbolic variables. The global functions are initialized to **1** (the Boolean function that is always true). STE updates the net values and the Boolean functions for each state vertex in the single sequence trajectory assertion. Line 10 computes the excitation for the circuit and stores it in the temporary variable $\tilde{Y}$. The use of $Sim(\tilde{Q})$ represents that the simulator is used to obtain the excitation for the circuit. Line 11 computes the defining trajectory label as the least upper bound of the excitation and the action lattice element, and updates the net values with the defining trajectory label. The function OK_A on line 12 maintains the condition under which the excitation is compatible with the action lattice element. As an example, assume that the simulator computes the excitation for a net to be the symbolic value u and the action node formula specifies that the same net should be asserted to the symbolic value v. The function OK_A would be $u \oplus v$ and the net value would be $(u \oplus v)\ ?\ u : D$. The function OK_R maintains the condition under which the reaction node formulas are satisfied. Finally, STE computes the OK function which

informally says that the trajectory assertion holds for the circuit if either the action node formulas are unsatisfiable or the reaction node formulas are satisfied.

```
verify(G) {
    Q̄ = X^m
    OK_A = 1
    OK_R = 1
    STE_Sequence(G)
    OK = ¬OK_A + OK_R
}
STE_Sequence(G) {
    for (i = 1 to k) {
        Ỹ = Sim(Q̃)                                  ..... Line 10
        Q̃ = Ỹ ⊔ Lat(σ_a(v_i))                       ..... Line 11
        OK_A = OK_A · [Ỹ ~ Lat(σ_a(v_i))]            ... Line 12
        OK_R = OK_R · [Q̃ ⊒ Lat(σ_r(v_i))]
    }
}
```

Figure 5.10: The STE algorithm for single sequence trajectory assertions.

An interesting property of the single sequence algorithm is that the symbolic computation is performed on only those variables that appear in the trajectory assertion. Unlike other symbolic circuit verifiers[6], STE does not introduce extra variables to represent the initial state or external input values.

5.5.3 Extensions for Acyclic Trajectory Assertion

An acyclic trajectory assertion can be verified by enumerating all the paths from source to sink and using STE separately on each path. There can, however, be an exponential number of paths in a graph. We use path variables to encode the graph and verify all paths in one single run of the simulator. The algorithm for encoding an acyclic trajectory assertion is shown in Figure 5.11. The encoding is specified in terms of a Boolean expression, $\mathcal{P}ath(w)$, defined over the path variables, which specifies all possible conditions under which there exists a path from the source to the vertex w. The encoding is performed in topological order of the vertices. Line 5 creates $\lceil \log(d(w)) \rceil$ number of variables for every vertex w with outdegree $d(w)$. Line 8 accumulates the path expressions on the vertices to compute the path function.

```
encodeGraph(G) {
    foreach (w ∈ W) Path(w) = 0
    Path(s) = 1
    foreach w ∈ W in topological order {
        Generate ⌈log(d(w))⌉ new variables ..... Line 5
        foreach ( (w, w_i) ∈ E ) {
            Let Enc(w, w_i) represent encoded expression for edge
            Path(w_i) = Path(w_i) + Path(w) · Enc(w, w_i) .... Line 8
        }
    }
}
```

Figure 5.11: Algorithm for encoding an acyclic trajectory assertion.

Consider the control graph shown in Figure 5.12. The edges are associated with encoded expressions. As an example, two path variables are created at the source to encode the three outgoing edges. The encoded expression associated with the edge from the source to vertex A is $Enc(\texttt{start}, \texttt{A}) = \overline{p_0} \cdot \overline{p_1}$. The vertices are associated with the path function. As an example $Path(\texttt{E}) = \overline{p_1} \cdot p_2 + p_0$. An assignment to the path variables to satisfy this function indicates all the paths from source to vertex E.

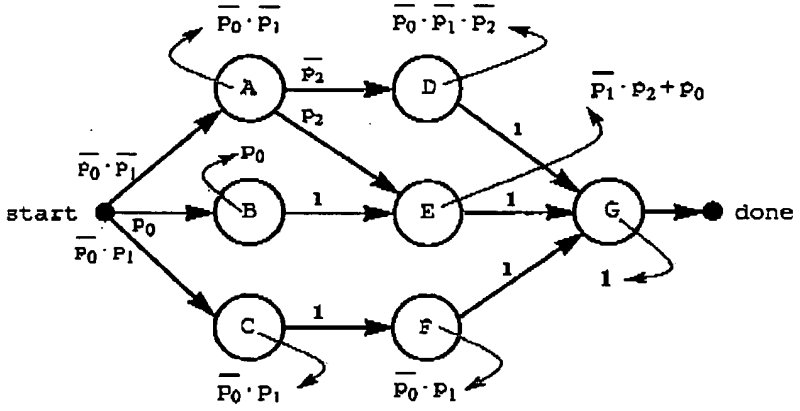


Figure 5.12: Symbolic Encoding of acyclic trajectory assertions.

STE has been extended with a conditional simulation capability. Conditional simulation allows STE to compute the next state under some specified Boolean condition. Under the condition, the simulator computes the next state. Under the complement of the condition, the simulator maintains

the previous state of the circuit. The extensions to the STE algorithm to verify acyclic trajectory assertions is shown in Figure 5.13. The routine **STE_Sequence()** presented earlier should be replaced by the routine **STE_Acyclic()**. The routine **STE_Acyclic()** computes the vertices in topological order. A topological order for the graph in Figure 5.12 is `start, A, D, B, E, C, F, G, done`. The path function serves as constraints for updating the net values, OK_A and OK_R.

STE_Acyclic(G) {
  foreach state vertex v in topological order {
    $\tilde{Y} = Sim(\tilde{Q})$
    $\tilde{Q} = \mathcal{P}ath(v)\ ?\ \tilde{Y} \sqcup \mathcal{L}at(\sigma_a(v)) : \tilde{Q}$
    $OK_A = OK_A \cdot [\overline{\mathcal{P}ath(v)} + [\tilde{Y} \sim \mathcal{L}at(\sigma_a(v))]]$
    $OK_R = OK_R \cdot [\overline{\mathcal{P}ath(v)} + [\tilde{Q} \sqsupseteq \mathcal{L}at(\sigma_r(v))]]$
  }
}

Figure 5.13: The STE algorithm for acyclic trajectory assertions.

Apart from serving as constraints for conditional simulation, the path variables serve another useful purpose. In effect, the above algorithm is performing a greatest lower bound computation at all vertices with multiple incoming edges. The path variables enable a *symbolic* greatest lower bound operation, thus avoiding the pessimism with the greatest lower bound operation on a pure scalar ternary model.

The symbolic computation in the acyclic version of STE is performed on both the variables that appear in the trajectory assertion and the path variables. Since the path variables are used to encode the trajectory assertion, the total number of variables that are symbolically manipulated is still determined solely by the trajectory assertion. The number of additional path variables introduced is $\sum_{w \in W} \lceil \log(d(w)) \rceil$.

5.5.4 Extensions for Generalized Trajectory Assertion

Seger and Bryant extended STE to deal with single sequence trajectory assertions augmented with a limited set of loops[18]. In particular, they could deal with a loop with a linear sequence of vertices as shown in Figure 5.14. The cycle $v_k, v_1, \ldots, v_i, \ldots, v_{k-1}, v_k$ represents a loop with a linear sequence of vertices.

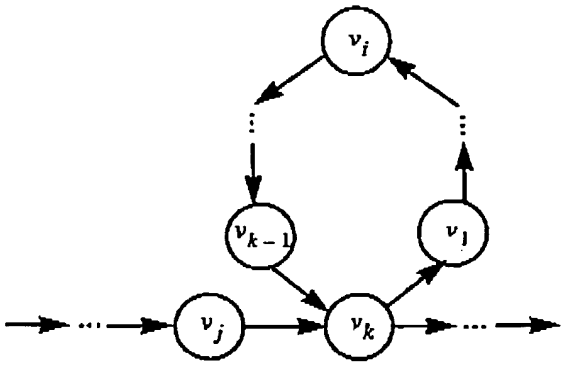


Figure 5.14: A loop with a linear sequence of vertices.

Assume that $\tilde{Q}$ represents the entry-level defining trajectory label on vertex v_k due to the edge (v_j, v_k). The algorithm for computing the defining trajectory label for vertex v_k due to the single sequence loop SSL is shown in Figure 5.15. The graph SSL^B, in line 5, is the single sequence trajectory assertion, shown in Figure 5.9, obtained after breaking the cycle in SSL. The algorithm computes the greatest fixed point of the recursive equation $\tilde{Q}^{i+1} = \tilde{Q}^i \sqcap \textbf{STE\_Sequence}(SSL^B)$. The routine **STE_Sequence()** was defined in Figure 5.10.

```
GFP_Sequence(SSL) {
   P̃ = ⊤
   while (P̃ ≠ Q̃) {
      P̃ = Q̃
      STE_Sequence(SSL^B)          ..... Line 5
      Q̃ = P̃ ⊓ Q̃
   }
}
```

Figure 5.15: Computing greatest fixed point for single sequence loops.

We have extended STE to deal with generalized trajectory assertions. Our STE algorithm to verify generalized trajectory assertions is shown in Figure 5.16. Line 2 identifies the strongly connected components in the trajectory assertion. A strongly connected component is a maximal set of vertices $V'' \subseteq V$ such that for every pair of vertices $v_1 \in V''$ and $v_2 \in V''$, there exists a path from v_1 to v_2 and a path from v_2 to v_1. In other words, vertices v_1 and v_2 are reachable from each other. The strongly connected components in the trajectory assertion can be identified with a linear time

$\Theta(W+E)$ algorithm[89]. Let G^{AC} represent the acyclic component graph obtained by shrinking each strongly connected component of G to a single vertex. The graph G^{AC} is an acyclic graph that can be processed using the acyclic version of STE. Line 4 encodes the acyclic graph G^{AC}. The vertices in these graph are traversed in topological order. Line 7 performs the greatest fixed point computation, if the vertex v is a representative vertex for a strongly connected component. Otherwise, line 9 updates the net values and global Boolean functions for vertex v in the acyclic graph.

```
STE_Generalized(G) {
    identifySCCs(G) ..... Line 2
    Let G^AC represent the acyclic component graph
    encodeGraph(G^AC) ..... Line 4
    foreach state vertex v in G^AC in topological order {
        if (v is representative vertex for SCC)
            GFP_Generalized(SCC) ..... Line 7
        else
            Update Q̃, OK_A, OK_R for vertex v ..... Line 9
    }
    quantifyPathVariables(G^AC) ..... Line 11
}
GFP_Generalized(SCC) {
    foreach entry point v_e into SCC {
        P̃ = ⊤
        Update Q̃, OK_A, OK_R due to external incoming edges into v_e.. ..... Line 16
        while (P̃ ≠ Q̃) { ..... Line 17
            P̃ = Q̃
            STE_Generalized(SCC^B)
            Q̃ = P̃ ⊓ Q̃
        } ..... Line 21
        foreach exit point v_x from SCC
            Update Q̃, OK_A, OK_R from v_e to v_x ..... Line 23
    }
}
```

Figure 5.16: STE algorithm for generalized trajectory assertions.

The routine **GFP_Generalized()** computes the greatest fixed point for every entry point into a strongly connected component. Consider a strongly connected component SCC with an entry

point v_e as shown in Figure 5.17. Line 16 updates the net values and global Boolean functions for all the incoming edges from the graph T_A. Lines 17-21 recursively computes the greatest fixed point of the equation: $\bar{Q}^{i+1} = \bar{Q}^i \sqcap$ **STE_Generalized**(SCC^B). The graph SCC^B is obtained from breaking cycles in the strongly connected component as shown in Figure 5.18. Line 23 walks from the entry point to each exit point for the strongly connected component and updates the net values and global Boolean functions.

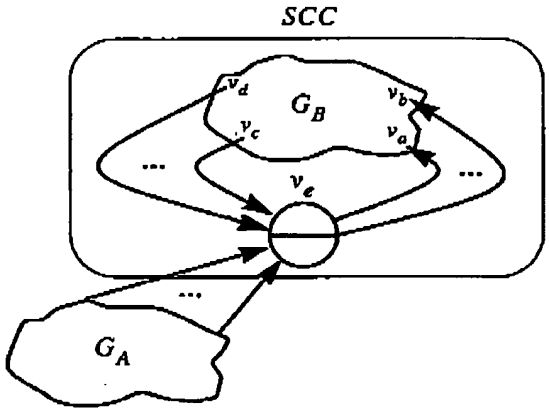


Figure 5.17: A strongly connected component.

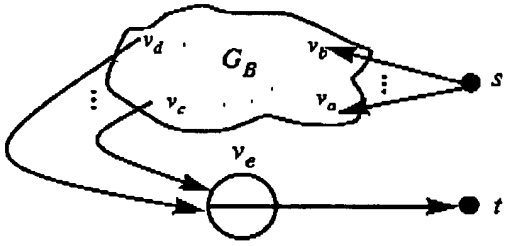


Figure 5.18: Breaking cycles in the strongly connected component.

An exact fixed point computation would require encoding the graph SCC^B with new path variables for each iteration in the algorithm until the fixed point was reached. In that case, the number of path variables would no longer be solely determined by the trajectory assertion. Therefore, line 11 uses the ternary quantification operation to quantify out the path variables introduced to encode the graph. The same path variables can now be reused in the next iteration.

Consider the generalized trajectory assertion shown in Figure 5.19. The algorithm will first identify the strongly connected component shown in the shaded box. The strongly connected component will be replaced by a single vertex to give the acyclic component graph shown in Figure 5.12, where the vertex E is the representative vertex for the strongly connected component. The acyclic component graph can be encoded as shown in Figure 5.12.

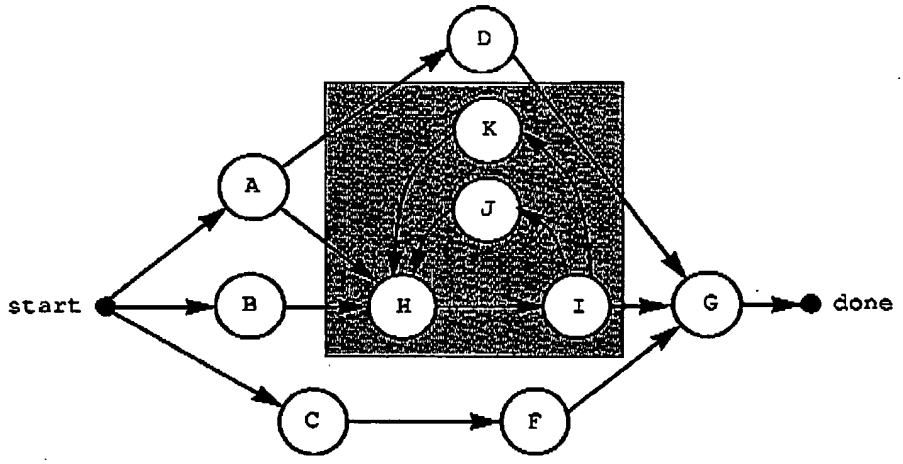


Figure 5.19: An example of a generalized trajectory assertion.

The graph obtained after breaking the cycles in the strongly connected component is shown in Figure 5.20. STE will perform the greatest fixed computation for this graph. The path variable p_3 is introduced to encode the graph. The variable will be quantified out at the end of each iteration of the fixed point computation and re-used in the next iteration.

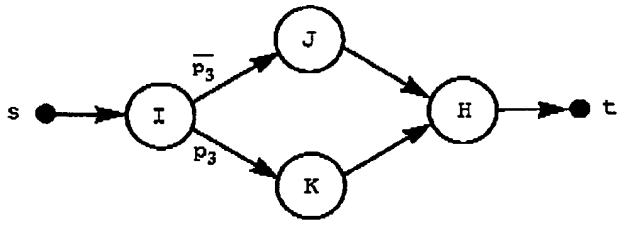


Figure 5.20: Breaking cycles in our example.

The ternary existential quantification at the end of each iteration implies that STE performs a conservative approximation of the reachability set. As in lattice-based trajectory checking, the conservative approximation can lead to a pessimistic verification. A pessimistic verification can generate a false negative, i.e., a correct circuit can be reported to be incorrect.

5.6 Summary

This chapter has presented Symbolic Trajectory Evaluation as a technique to verify trajectory assertions on the actual circuit design. The main strength of STE is that it avoids the need to compute the next-state function for the entire circuit.

The previous four chapters have introduced the main components of our verification methodology. What remains to put this all together into a methodology is the theory. The next chapter defines the theoretical foundation for our methodology for formal hardware verification using Symbolic Trajectory Evaluation.

Chapter 6

Putting It All Together

The last few chapters have revealed various aspects of our verification methodology. This chapter focuses on putting all these concepts together and developing a firm theoretical foundation for the formal verification of circuits using Symbolic Trajectory Evaluation. The chapter first reviews the concepts of the abstract specification and the implementation mapping. STE is used to verify that each abstract assertion individually holds for the circuit under some mapping. We, however, want to ensure that the entire abstract specification holds for the circuit. This chapter explores the question of what it means for the abstract specification to hold for the circuit under some mapping.

6.1 Abstract Specification

Chapter 2 introduced the form of the abstract specification. The specification is defined as a set of abstract assertions. Each abstract assertion is of the form: $A = P \Rightarrow Q$, where P and Q are abstract formulas. Abstract formulas are defined over a set of abstract elements S_v. Let S be the set of assignments to abstract elements, $S = \{0, 1\}^n$, where $n = |S_v|$.

The following definition and property of the satisfying set has been borrowed from Chapter 2.

Definition 1 (Satisfying Set): Let the satisfying set, written as $Sat(AF)$, for an abstract formula AF be the set of assignments to abstract elements that satisfy the abstract formula.

Property of Satisfying Set: $Sat(AF_1 \text{ and } AF_2) = Sat(AF_1) \cap Sat(AF_2)$.

In the remainder of this chapter, we will use the notation $\hat{AF}$ as a shorthand for $Sat(AF)$. We can now view an assertion as defining a transition on the set of abstract assignments. An assertion $P \Rightarrow Q$ can now be viewed as defining a transition from the set $\hat{P} = Sat(P)$ to the set $\hat{Q} = Sat(Q)$. In these terms, the meaning of an assertion is that $\forall(p \in \hat{P})$, $\exists(q \in \hat{Q})$, such that (p, q) is a transition in the abstract machine.

Definition 2 (Abstract Assertion): Let i index the set of abstract assertions. Now we have a set of abstract assertions $A_i = P_i \Rightarrow Q_i$. The set of abstract assertions can also be viewed as a set of tuples of the form: $\langle \hat{P}_i, \hat{Q}_i \rangle$.

Definition 3 (Abstract Specification): The abstract specification can be viewed to be a set of tuples of the form: $\langle \{p_j\}, \bigcap_{i \mid p_j \in \hat{P}_i} \hat{Q}_i \rangle$, where $p_j \in \bigcup_i \hat{P}_i$.

As an example, consider a system with two abstract elements A and B. Assume that the system has been specified with two abstract assertions:

- (A is 0) ➡ (A is 0)
- (B is 0) ➡ (B is 1) .

In the set model, these abstract assertions can be represented as $\{\langle\{00, 01\}, \{00, 01\}\rangle, \langle\{00, 10\}, \{01, 11\}\rangle\}$, where the left bit is the value of state A and right bit is the value of state B.

The abstract specification is the set $\{\langle\{00\}, \{01\}\rangle, \langle\{01\}, \{00, 01\}\rangle, \langle\{10\}, \{01, 11\}\rangle\}$. This corresponds to three mutually exclusive assertions:

- (A is 0) **and** (B is 0) ➡ (A is 0) **and** (B is 1)
- (A is 0) **and** (B is 1) ➡ (A is 0)
- (A is 1) **and** (B is 0) ➡ (B is 1) .

6.2 Trajectory Specification

Chapter 3 introduced the form of the implementation mapping. A map machine for a simple abstract formula s is a tuple of the form: $Map(s) = \langle V, U, E, s, \iota, \sigma_a, \sigma_r, \Upsilon \rangle$. The labellings σ_a and σ_r label state vertices with node formulas. Node formulas are defined over a set of node elements N_v. Let N be the set of assignments to node elements, $N = \{0, 1\}^m$, where $m = |N_v|$. Let $\mathcal{F}_{simple}$ represent the set of all simple abstract formulas. Since there are two simple abstract formulas associated with each abstract element, $|\mathcal{F}_{simple}| = 2n$.

Chapter 4 gave an overview of the trajectory generation phase. The following definitions and properties of the trajectory formula and trajectory assertion are borrowed from Chapter 4.

Definition 4 (Trajectory Formula): The trajectory formula, written as $\mathcal{TF}(AF)$, defines a control graph with node formulas for an abstract formula AF.

Property of Trajectory Formula: $\mathcal{TF}(AF_1 \text{ and } AF_2) = \mathcal{TF}(AF_1) \,||\, \mathcal{TF}(AF_2)$.

Definition 5 (Trajectory Assertion): Given an abstract assertion, $A = P \rightarrow Q$, the corresponding trajectory assertion is defined to be $\mathcal{TA}(A) = \mathcal{TF}(P) \,//\, [\mathcal{TF}(Q)]^M$. The trajectory assertion $\mathcal{TA}(A)$ is a control graph with action and reaction node formulas. The most general form is a prescient trajectory assertion.

So far, we have viewed the mapping to be applied over abstract formulas. For the purposes of this chapter, it is useful to consider the mapping applied over sets of abstract assignments. We can view the user-specified map machine $\mathcal{M}ap(s)$ as a mapping J applied over $\mathcal{S}at(s)$ as follows:

Definition 6 (Set-level Mapping): Define a mapping J such that $\forall s \in \mathcal{F}_{simple}$, $J(\hat{s}) = \mathcal{M}ap(s)$, where $\hat{s} = \mathcal{S}at(s)$

Given the mapping over the simple abstract formulas, the mapping over other subsets of S can be derived using the following property:

Property of Set-level Mapping: $J(\hat{X} \cap \hat{Y}) = J(\hat{X}) \,||\, J(\hat{Y})$ where $\hat{X} \subseteq S$ and $\hat{Y} \subseteq S$.

A precondition P can be viewed to be of the form $P = P^S \text{ and } P^D$, where P^S is the conjunction and domain restriction of simple abstract formulas that have single-sided map machines and P^D is the conjunction and domain restriction of simple abstract formulas that have double-sided map machines[1]. In terms of sets of abstract assignments this implies that $\mathcal{S}at(P) = \mathcal{S}at(P^S) \cap \mathcal{S}at(P^D)$ which translates to $\hat{P} = \hat{P^S} \cap \hat{P^D}$ in our shorthand notation.

1. As an aside, note that an abstract formula of the form $e \rightarrow (P^S \text{ and } P^D)$ can be written as $(e \rightarrow P^S) \text{ and } (e \rightarrow P^D)$.

Let us now consider a set of abstract assertions $\langle \hat{P}_i, \hat{Q}_i \rangle$, where $\hat{P}_i = \hat{P}_i^S \cap \hat{P}_i^D$. We place the restriction that for any two assertions j and k, either $\hat{P}_j^D \cap \hat{P}_k^D = \varnothing$ or $\hat{P}_j^D = \hat{P}_k^D$. It is the task of the user to ensure that the set of assertions obey this restriction. We shall refer to this restriction as the *double-sided restriction*. The reason to impose this double-sided restriction on the abstract specification will be made apparent later on in this chapter. The intuition is that double-sided map machines in the precondition relate abstract inputs into a set of bidirectional circuit signals. The reaction node formulas in these map machines define a set of allowed responses on circuit outputs. If the abstract input does not appear in the precondition, STE will not check the response on the circuit output and an incorrect circuit may be reported to be correct.

6.3 Verification of an Abstract Assertion

Chapter 5 described how to use STE to verify trajectory assertions on a circuit model. Chapter 5 concentrated on describing an algorithm to verify trajectory assertions. Here we attempt to obtain a deeper understanding of what it means to verify a trajectory assertion.

Let us first define a model for the circuit. Chapter 5 introduced a circuit excitation function, $\eta : N \rightarrow 2^N$. The circuit can be modeled as a set of trajectories.

Definition 7 (Set of Trajectories): Define an infinite trajectory τ as $\tau = \tau_1\tau_2\tau_3\ldots$, where $\tau_i \subseteq N$ and $\forall_i \tau_i \in \eta(\tau_{i-1})$. And let T denote the set of all trajectories associated with a circuit.

Let us first consider the meaning of a trajectory assertion $G = \mathcal{TA}(A)$. And for the moment, we will consider the most general form of the trajectory assertion, i.e., a prescient trajectory assertion.

Definition 8 (Meaning of a Trajectory Assertion): A prescient trajectory assertion G divides the set of trajectories into 3 separate sets, i.e., the accept, reject, and don't care sets. The definition of these sets requires some additional machinery. A path p in the trajectory assertion G is a path from source to sink. The length function, $len(p)$, denotes the number of state vertices in the path. Therefore a path of length m is of the form $p = s, v_1, \ldots, v_m, t$ where $v_i \in V$. Define an upto function that takes in a circuit trajectory and a path and returns an integer. The function $upto(\tau, p)$ returns the integer k such that $\forall_{i=1}^{k} \tau_i \in Set(\sigma_a(v_i)) \cap Set(\sigma_r(v_i))$ and either

$k = len(p)$ or $\tau_{k+1} \notin Set(\sigma_a(v_{k+1})) \cap Set(\sigma_r(v_{k+1}))$. Now the accept, don't care, and reject sets can be defined as follows:

$$Acc(G,T) = \left\{ \tau \;\middle|\; \begin{array}{c} \tau \in T, \text{ and} \\ \exists \text{ path } p \text{ in } G,\ upto(\tau, p) = len(p) \end{array} \right\}$$

$$DC(G,T) = \left\{ \tau \;\middle|\; \begin{array}{c} \tau \in T, \text{ and} \\ \forall \text{ paths } p \text{ in } G,\ upto(\tau, p) = k, \\ k < len(p), \\ \tau_{k+1} \notin Set(\sigma_a(v_{k+1})) \end{array} \right\}$$

$$Rej(G,T) = \left\{ \tau \;\middle|\; \begin{array}{c} \tau \in T, \text{ and} \\ \forall \text{ paths } p \text{ in } G,\ upto(\tau, p) < len(p), \text{ and} \\ \exists \text{ path } p \text{ in } G,\ upto(\tau, p) = k, \\ \tau_{k+1} \in Set(\sigma_a(v_{k+1})), \\ \tau_{k+1} \notin Set(\sigma_r(v_{k+1})) \end{array} \right\}$$

A trajectory is accepted if there exists a path from source to sink in the graph such that the trajectory satisfies the action and reaction node formulas along the path. A trajectory is a don't care if there does not exist a path in the graph such that the trajectory satisfies the action node formulas along the path. A trajectory is rejected if it is not accepted and not a don't care, i.e. there does not exist a path such that the trajectory satisfies the action and reaction node formulas and there does exist a path such that the action node formulas are satisfied but the reaction node formulas are not satisfied.

Definition 9 (Verification of an Abstract Assertion): Consider an abstract assertion $\langle \hat{P}, \hat{Q} \rangle$ and a circuit with a set of trajectories T. For the abstract assertion to hold for the circuit under a mapping J, the verification task has to show that $Rej[J(\hat{P}) \,//\, (J(\hat{Q}))^M, T] = \varnothing$.

The construction $G = J(\hat{P}) \,//\, [J(\hat{Q})]^M$ is in general a prescient trajectory assertion. Unfortunately, it seems that a verification algorithm for a prescient trajectory assertion would not only need to construct the set T for circuit, but also enumerate each path in the trajectory assertion. This would be prohibitively expensive. On the other hand, if we restrict ourselves to oblivious trajectory assertions, then we can define verification algorithms that do not have to explicitly con-

struct the set T and do not have to explicitly enumerate each path in the graph. This is the reason that in Chapter 5, we limited ourselves to oblivious trajectory assertions.

Some of the relevant properties for the accept, reject, and don't care sets are given below.

Property of Mirror Operation: $Acc(G^M, T) = Acc(G, T)$, where G^M is the mirror of G obtained by reversing the roles of the action and reaction node formulas.

Properties of Compose Operation: We can define the accept, reject, and don't care sets for the parallel composition operation as follows:

- $Acc(G_1 \,||\, G_2, T) = Acc(G_1, T) \cap Acc(G_2, T)$
- $Rej(G_1 \,||\, G_2, T) = Rej(G_1, T) \cap Acc(G_2, T) \cup Acc(G_1, T) \cap Rej(G_2, T) \cup Rej(G_1, T) \cap Rej(G_2, T)$
- $DC(G_1 \,||\, G_2, T) = DC(G_1, T) \cup DC(G_2, T)$

6.4 Verification of the Abstract Specification

The above verification task has shown that each abstract assertion individually holds for the circuit. We, however, want to ensure that the entire abstract specification holds for the circuit. As an example, assume that the verification task has been used to verify that the abstract assertions (A is 0) ➡ (A is 0) and (B is 0) ➡ (B is 1) hold for the circuit under some user-defined implementation mapping. The abstract specification corresponding to these two assertions is defined as following three mutually exclusive assertions: (A is 0) **and** (B is 0) ➡ (A is 0) **and** (B is 1), (A is 0) **and** (B is 1) ➡ (A is 0) and (A is 1) **and** (B is 0) ➡ (B is 1). We want to ensure that the three mutually exclusive assertions hold for the circuit under the mapping. We now define what it means for the abstract specification to hold for the circuit.

Definition 10 (Verification of the Abstract Specification): Consider the set of abstract assertions $\langle \hat{P}_i, \hat{Q}_i \rangle$. We know that the corresponding abstract specification is the set $\langle \{p_j\}, \bigcap_{i \mid p_j \in \hat{P}_i} \hat{Q}_i \rangle$ where $p_j \in \bigcup_i \hat{P}_i$. For the abstract specification to hold for the circuit with a set of trajectories T under a mapping J, it has to be shown that $Rej\left[J\{p_j\} \,//\, J\left[\bigcap_{i \mid p_j \in \hat{P}_i} \hat{Q}_i\right]^M, T\right] = \varnothing$.

The verification task will ensure that each abstract assertion individually holds for the circuit. Mathematically speaking, that will ensure $\forall i,\ Rej[J(\hat{P}_i) // (J(\hat{Q}_i))^M, T] = \varnothing$. The verification of the abstract specification will follow from the verification of the individual abstract assertions if the following properties hold:

- **Antecedent Property:** $Rej[J(\hat{P}_1 \cap \hat{P}_2) // (J(\hat{Q}))^M, T] \subseteq Rej[J(\hat{P}_1) // (J(\hat{Q}))^M, T]$ when $\hat{P}_1 \cap \hat{P}_2 \neq \varnothing$.
- **Consequent Property:** $Rej[J(\hat{P}) // (J(\hat{Q}_1))^M, T] = \varnothing$ and $Rej[J(\hat{P}) // (J(\hat{Q}_2))^M, T] = \varnothing$ implies $Rej[J(\hat{P}) // (J(\hat{Q}_1 \cap \hat{Q}_2))^M, T] = \varnothing$.

Consider two abstract assertions, $P_1 \Rightarrow Q$ and $(P_1 \text{ and } P_2) \Rightarrow Q$. The antecedent property ensures that if the assertion $P_1 \Rightarrow Q$ holds for the circuit under some mapping, then the assertion $(P_1 \text{ and } P_2) \Rightarrow Q$ also holds for the circuit. In other words, if the mapping of the set $\hat{P}_1$ leads to an element in the mapping of the set $\hat{Q}$, then the mapping of a *subset* of $\hat{P}_1$ should also lead to an element in the mapping of set $\hat{Q}$.

Consider two abstract assertions, $P \Rightarrow Q_1$ and $P \Rightarrow Q_2$. The consequent property ensures that if both assertions $P \Rightarrow Q_1$ and $P \Rightarrow Q_2$ hold for the circuit under some mapping, then the assertion $P \Rightarrow (Q_1 \text{ and } Q_2)$ also holds for the circuit. In other words, if the mapping of the set $\hat{P}$ leads to an element in the mapping of both sets $\hat{Q}_1$ and $\hat{Q}_2$, then the mapping of $\hat{P}$ should also lead to an element in the mapping of the set $\hat{Q}_1 \cap \hat{Q}_2$.

The next two sections provide a proof for these properties. As will be seen, the double-sided restriction on the set of abstract assertions is required to prove the antecedent property.

6.4.1 Antecedent Property

Proof: Consider the set of assertions $\{\langle \hat{P}_1, \hat{Q} \rangle, \langle \hat{P}_2, \hat{Q} \rangle\}$. We know that $\hat{P}_1 = \hat{P}_1^S \cap \hat{P}_1^D$ and $\hat{P}_2 = \hat{P}_2^S \cap \hat{P}_2^D$. From the double-sided restriction on the set of assertions we have that either $\hat{P}_1^D \cap \hat{P}_2^D = \varnothing$ or $\hat{P}_1^D = \hat{P}_2^D$. Since we want to consider the case only when $\hat{P}_1 \cap \hat{P}_2 \neq \varnothing$, let us assume that $\hat{P^D} = \hat{P}_1^D = \hat{P}_2^D$.

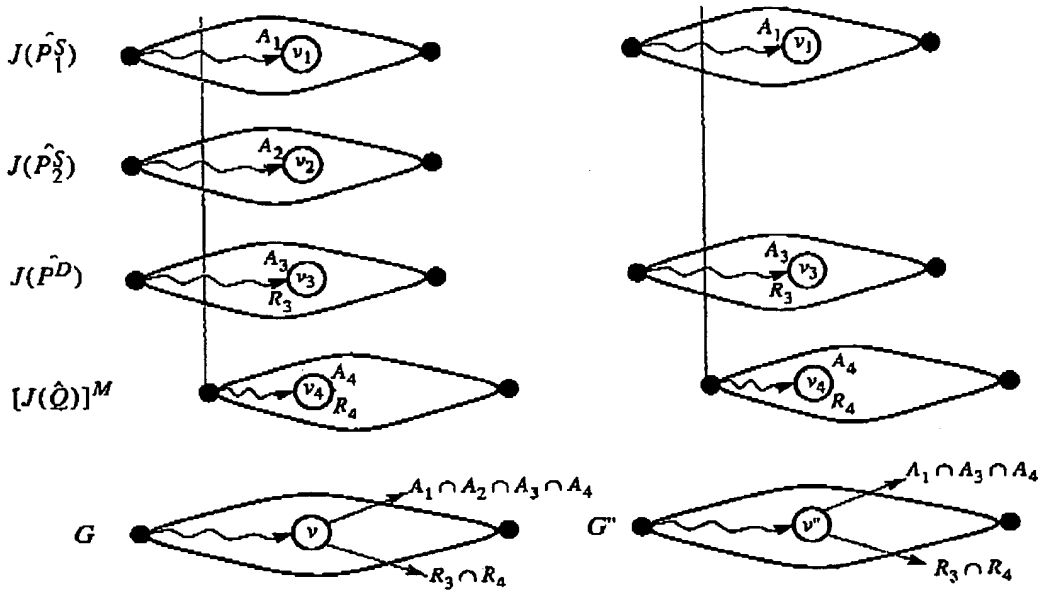


Figure 6.1: Construction of graphs G and G".

The trajectory assertion $J(\hat{P}_1 \cap \hat{P}_2) /\!/ (J(\hat{Q}))^M = [J(\hat{P}_1^S) \,||\, J(\hat{P}_2^S) \,||\, J(\hat{P}^D)] /\!/ (J(\hat{Q}))^M$. We shall refer to this trajectory assertion as G. Consider an infinite trajectory $\tau \in Rej(G, T)$. Since τ is rejected there exists some vertex v in G and an index k such that $\tau_k \in A$ and $\tau_k \notin R$ where A and R are set of action and reaction node assignments associated with vertex v. In other words $A = Set(\sigma_a(v))$ and $R = Set(\sigma_r(v))$. Assume that the vertex v is the result of composition of vertices v_1 in $J(\hat{P}_1^S)$, v_2 in $J(\hat{P}_2^S)$, v_3 in $J(\hat{P}^D)$ and v_4 in $[J(\hat{Q})]^M$ as shown in Figure 6.1. Let A_1 and A_2 be the set of action node assignments associated with vertex v_1 and v_2 respectively. Since $J(\hat{P}_1^S)$ and $J(\hat{P}_2^S)$ are single-sided mappings, we know that the reaction node assignments is the set N. Let A_3 and R_3 be the action and reaction node assignments associated with vertex v_3. And let A_4 and R_4 be the action and reaction node assignments associated with vertex v_4. Since v is the result of composing vertices v_1 to v_4, we have that $A = A_1 \cap A_2 \cap A_3 \cap A_4$ and $R = R_3 \cap R_4$. Therefore, $\tau_k \in A_1 \cap A_2 \cap A_3 \cap A_4$ and $\tau_k \notin R_3 \cap R_4$.

The trajectory assertion $J(\hat{P}_1) /\!/ [J(\hat{Q})]^M = [J(\hat{P}_1^S) \,||\, J(\hat{P}^D)] /\!/ [J(\hat{Q})]^M$. We shall refer to this graph as G''. We know that there exists a vertex v'' in G'' that is the composition of vertices v_1 in $J(\hat{P}_1^S)$, v_3 in $J(\hat{P}^D)$ and v_4 in $[J(\hat{Q})]^M$. Therefore, the action node assignment associ-

ated with v" is $A" = A_1 \cap A_3 \cap A_4$. The reaction node assignment associated with vertex v" is $R" = R_3 \cap R_4$. Therefore, we know that $\tau_k \in A"$ and $\tau_k \notin R"$. This implies that $\tau \in Rej(G", T)$ too. In other words, any trajectory rejected by the graph G has to be rejected by the graph $G"$. This completes the proof.

The safest way to impose the double-sided restriction property is for the user to ensure that abstract elements with double-sided map machines appear in each and every abstract assertion. Let us consider an example to shown the importance of this property.

Consider an abstract system with abstract elements A, B, and C. Assume that the abstract system performs the conjunction of the inputs, A and B, to generate the output C. Assume an implementation of the abstract system that uses interface protocols to obtain the A and B operands as shown in Figure 6.2. The protocol for fetching the A and B operand is implemented with Moore machines. Consider the A operand. The data is received when the rdyA signal goes high. On an active rdyA signal the machine transitions to state S2 and acknowledges the data. A similar case can be made for the B operand. However, let us assume that there is a bug in the circuit, so that the B data is never acknowledged. In other words, the ackB signal remains at logic 0 for both states S3 and S4.

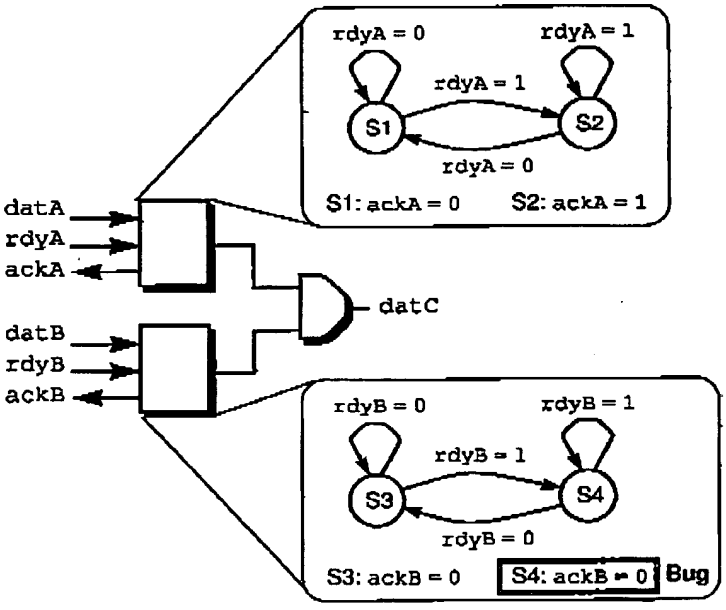


Figure 6.2: An implementation for our example system.

Further, assume that the user provides the double-sided map machines for the A and B operands as shown in Figure 6.3.

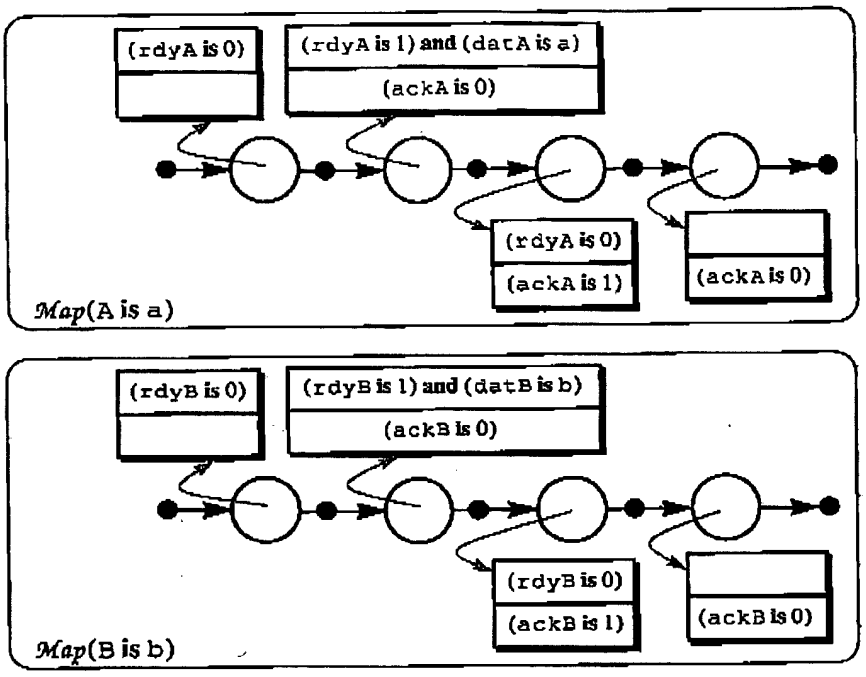


Figure 6.3: The implementation mapping for our example.

For the moment, let us ignore the double-sided restriction property. Consider that the user specified the abstract assertion (A is 0) ➡ (C is 0) . STE can be used to show that the abstract assertion holds for the circuit under the given implementation mapping. Having shown that the more general abstract assertion, (A is 0) ➡ (C is 0) , holds for the circuit, we would like to infer that the more specific abstract assertions, (A is 0) **and** (B is 0) ➡ (C is 0) and (A is 0) **and** (B is 1) ➡ (C is 0) , will also hold for the circuit. We, however, know that this is not true. The specific abstract assertions will be rejected by STE due to the check for the ackB signal.

The reason for this problem is that the user ignored the double-sided restriction property. The abstract assertion (A is 0) ➡ (C is 0) is not a valid assertion. Both A and B have double-sided mappings. Therefore, both A and B have to appear in each and every abstract assertion.

6.4.2 Consequent Property

Proof by Contradiction: Assume that $Rej[J(\hat{P}) // (J(\hat{Q}_1))^M, T] = \varnothing$ and $Rej[J(\hat{P}) // (J(\hat{Q}_2))^M, T] = \varnothing$, and $Rej[J(\hat{P}) // (J(\hat{Q}_1 \cap \hat{Q}_2))^M, T] \neq \varnothing$.

The trajectory assertion $J(\hat{P}) // [J(\hat{Q}_1 \cap \hat{Q}_2)]^M = J(\hat{P}) // [J(\hat{Q}_1) \,||\, J(\hat{Q}_2)]^M$. We shall refer to this trajectory assertion as G. There is at least one infinite trajectory $\tau \in Rej(G, T)$. Since τ is rejected there exists some vertex v in G and an index k such that $\tau_k \in A$ and $\tau_k \notin R$ where A and R are set of action and reaction node assignments associated with vertex v. Assume that the vertex v is the result of composition of vertices v_1 in $J(\hat{P})$, v_2 in $[J(\hat{Q}_1)]^M$, and v_3 in $[J(\hat{Q}_2)]^M$. Let A_1 and R_1 be the set of action and reaction node assignments associated with vertex v_1. Let A_2 and R_2 be the action and reaction node assignments associated with vertex v_2. And let A_3 and R_3 be the action and reaction node assignments associated with vertex v_3. Since v is the result of composing vertices v_1, v_2 and v_3, we have that $A = A_1 \cap A_2 \cap A_3$ and $R = R_1 \cap R_2 \cap R_3$. Therefore $\tau_k \in A_1 \cap A_2 \cap A_3$ and $\tau_k \notin R_1 \cap R_2 \cap R_3$.

Consider the trajectory assertion $J(\hat{P}) // [J(\hat{Q}_1)]^M$. We shall refer to this graph as G''. We know that there exists a vertex v'' in G'' which is the composition of vertices v_1 in $J(\hat{P})$ and v_2 in $[J(\hat{Q}_1)]^M$. Therefore, the action node assignment associated with v'' is $A'' = A_1 \cap A_2$. The reaction node assignment associated with vertex v'' is $R'' = R_1 \cap R_2$. Since $Rej(G'', T) = \varnothing$, the trajectory τ could belong either to the accept or don't care set. Since we know that $\tau_k \in A_1 \cap A_2$, the trajectory belongs to the accept set and $\tau_k \in R_1 \cap R_2$.

Similarly, from trajectory assertion $J(\hat{P}) // [J(\hat{Q}_2)]^M$, we can obtain that $\tau_k \in A_1 \cap A_3$ and $\tau_k \in R_1 \cap R_3$.

But now we have that $\tau_k \notin R_1 \cap R_2 \cap R_3$ and $\tau_k \in R_1 \cap R_2$ and $\tau_k \in R_1 \cap R_3$. This is a contradiction. So our assumption is incorrect. This completes the proof by contradiction.

6.5 Summary

The languages and tools were used to verify that each individual abstract assertion holds for the circuit under the implementation mapping. The theory made the claim that the entire abstract specification holds for the circuit under the mapping. This required the user to ensure the double-sided restriction property on the abstract specification and implementation mapping.

Chapter 7

Reasoning About Execution Sequences

Once we have verified each individual operation on a circuit, we want to reason about infinite execution sequences. This chapter describes some of our preliminary work in the creation of execution sequences.

The concept of the main machine was introduced in Chapter 3. The chapter informally introduced the concept of composition to stitch main machines together to form execution sequences. An infinite execution sequence, however, required the composition of infinite copies of the main machine. This chapter introduces the concept of a closure construction as a means to stitch machines together to form all feasible execution sequences.

7.1 Related Work

Beatty reduced the task of verification for all possible execution sequences into a check for three separate properties, i.e., *Obedience*, *Conformity* and *Distinction*[13][15]. The Obedience property was the check to verify that the trajectory assertion holds for the circuit. The Conformity property required that for every specification input sequence, there should be a corresponding circuit input sequence. The Distinction property required that two distinct specification output sequences should have distinct circuit output sequences. Beatty was able to claim that the conformity check needs to be performed only on the abstract inputs. Internal state elements do not need to be checked for conformity since they appear on both sides of the assertion.

So far, this thesis has dealt with the Obedience property. This chapter is the first step towards reasoning about execution sequences. It remains to be seen if Obedience, Conformity, and Distinction represent the complete set of properties to verify all possible execution sequences in our extended framework.

7.2 Main Machine

Section 3.3.2 defined the main machine to be a control graph with a nextmarker function. The main machine defines the flow of control for an arbitrary i^{th} system operation. We can make this explicit by associating an operation index counter with the main machine. In this discussion, the operation index will be specified as a superscript on the set of vertices. Therefore, a state vertex v^i refers to a state vertex for the i^{th} operation. The main machine can be defined as follows:

Definition 1 (Main Machine): Define M^i to be the main machine for the i^{th} operation, where the superscript i refers to the operation index counter. The main machine is of the form: $M^i = \langle V^i, U^i, E^i, s^i, t^i, \beta^i \rangle$, where

- $\langle V^i, U^i, E^i, s^i, t^i \rangle$ is a control graph.
- β^i is the nextmarker function, $\beta^i : U^i \rightarrow U^i$.

The main machine defines not only the flow of control for individual system operations, but also defines how operations can be stitched together to form execution sequences. An operation-level view of the system should not only define the flow of control i^{th} operation, but all subsequent operations. An operation-level view of the pipeline can be obtained by augmenting our main machine with an *incrementing edge*, $(\beta^i(s^i), s^i)$. The incrementing edge is a directed edge from the time point where the next operation can start to the source of the main machine. A traversal through the incrementing edge increments the operation index counter by 1. The operation-level view of main machine is shown in Figure 7.1. The incrementing edges are shown as thick dashed edges.

Figure 7.1: Operation-level view of main machine.

As an example, consider a simple processor with fetch, decode, and execute pipeline stages. Assume that all instructions spend a single clock cycle in each pipeline stage. The operation level view of the main machine for this processor is shown in Figure 7.2. The main machine can be

interpreted as follows: Assume that the operation index counter i is currently 1, indicating that we are processing the first operation. The state vertex F^1 represents the fetch stage for the first operation. At the completion of F^1, we can advance to state vertex D^1 indicating the decode stage of the first operation. In addition, we can traverse the incrementing edge which increments the operation index counter i to 2, and advance to state vertex F^2. The state vertex F^2 represents the fetch stage of the second operation. Thus the decode stage of the first operation overlaps with the fetch stage of the second operation. It can be seen that this representation succinctly captures the operation-level view of the pipeline.

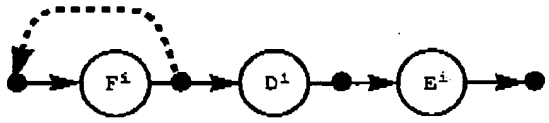


Figure 7.2: Example 1. Operation-level view of main machine for simple processor.

7.3 Closure Construction

It is possible to obtain a closed form M^* of the main machine, which represents all possible execution sequences. The main machine can be viewed as describing the operation-level view of the machine, and the closed machine M^* can be viewed as describing the system-level view of the pipeline.

Let us define M^{k*} as the machine obtained from composing k main machines $M^i, M^{i+1}, \ldots, M^{i+k-1}$. For the moment, we will informally define M^{k*}. Later on, we will give a more rigorous mathematical definition. The result of composing $k+1$ main machines can be defined as $M^{k+1*} = M^{k*} // M^{i+k}$. The operator // is the shift-and-compose operator described in Chapter 4. Consider that M^{k*} is associated with two cutsets, i.e., the A and B-cutsets. The A-cutset represents the time point where the next operation M^{i+k} can start. The B-cutset represents the time point where the first operation M^i ends. Figure 7.3 shows the effect of composition on the A and B-cutsets. The A-cutset in machine M^{k+1*} corresponds to the time point where the next operation can start in the main machine M^{i+k}. The B-cutset in machine M^{k+1*} corresponds to the B-cutset in machine M^{k*}.

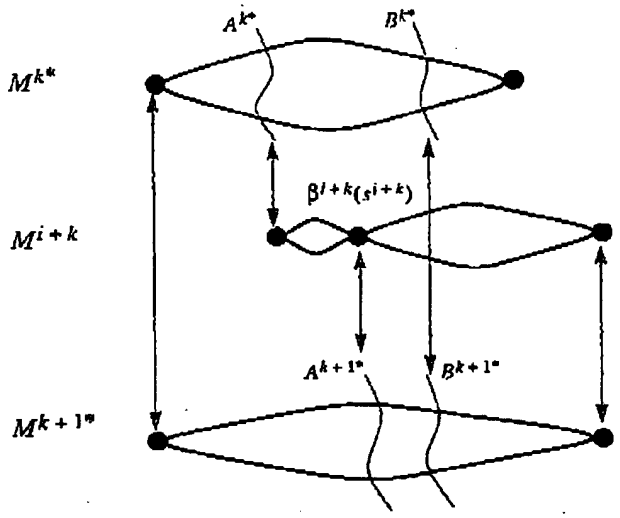


Figure 7.3: Composing main machines.

We will keep on composing main machines in this manner until the A-cutset moves beyond the B-cutset. Since the B-cutset corresponds to the end of the first operation, this implies that the first operation M^i has successfully completed. For serial pipelines with d pipeline stages, this should happen after composing $d+1$ main machines. Therefore, in machine M^{d+1*}, the cutset A^{d+1*} should have moved beyond the cutset B^{d+1*}. In machine M^{d+1*}, we can use a normalization operation to replace all transitions beyond the cutset B^{d+1*} by incrementing edges.

The closure construction is a two step process:

- Compose main machines.
- Normalize the resultant composed machine.

Details of the composition and normalization operation are presented in the next two sections.

7.3.1 Composing Main Machines

Let us first rigorously define the concept of a k-composed machine.

Definition 2 (k-Composed Machine): Define $M^{k*} = M^i // M^{i+1} // \ldots // M^{i+k-1}$. The k-composed machine M^{k*} is defined to be of the form: $M^{k*} = \langle V^{k*}, U^{k*}, E^{k*}, s^{k*}, t^{k*}, A^{k*}, B^{k*} \rangle$, where

- $\langle V^{k*}, U^{k*}, E^{k*}, s^{k*}, t^{k*} \rangle$ is a control graph.
- A^{k*} is a vertex cut of the control graph, $A^{k*} \subseteq U^{k*}$.
- B^{k*} is a vertex cut of the control graph, $B^{k*} \subseteq U^{k*}$.

The vertex cut A^{k*} represents a cutset of event vertices where the next operation, M^{i+k}, can be started. The vertex cut B^{k*} represents a cutset of event vertices that corresponds to the end of the first operation M^i.

The 1-composed machine M^{1*} is a trivial case and represents a sequence of just a single operation. The control graph for M^{1*} is defined to be the control graph for the main machine M^i. The A-cutset is defined to be the nextmarker of the source of machine M^i. The B-cutset is defined to be the sink of machine M^i. Thus $M^{1*} = \langle V^{1*}, U^{1*}, E^{1*}, s^{1*}, t^{1*}, A^{1*}, B^{1*} \rangle$, where

- $\langle V^{1*}, U^{1*}, E^{1*}, s^{1*}, t^{1*} \rangle = \langle V^i, U^i, E^i, s^i, t^i \rangle$.
- $A^{1*} = \{\beta^i(s^i)\}$.
- $B^{1*} = \{t^i\}$.

The k-composed machine can be iteratively constructed as $M^{k+1*} = M^{k*} // M^{i+k}$. The 1-composed machine M^{1*} will form the base case for this iteration. The start of machine M^{i+k} is shifted to the A-cutset of machine M^{k*}. The machines are augmented as shown in Figure 7.4.

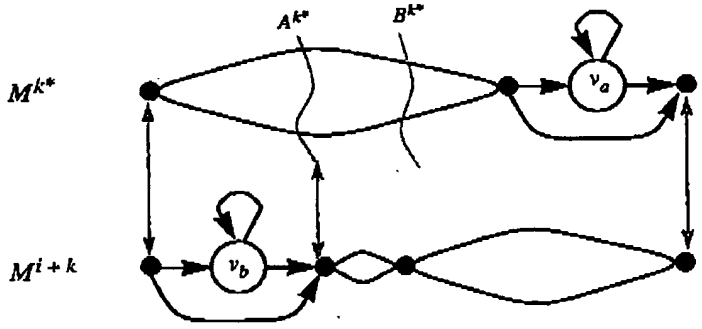


Figure 7.4: Augmenting main machines.

Now, we can take the composition of the two augmented machines. The composition will carefully preserve event vertices in M^{k+1*} that are due to the B-cutset in M^{k*} or due to the event vertex $\beta^{i+k}(s^{i+k})$ in M^{i+k}. Assume an augmented k-composed machine

$M^{k*} = \langle V^{k*}, U^{k*}, E^{k*}, s^{k*}, t^{k*}, A^{k*}, B^{k*} \rangle$ and an augmented main machine $M^{i+k} = \langle V^{i+k}, U^{i+k}, E^{i+k}, s^{i+k}, t^{i+k}, \beta^{i+k} \rangle$. Both machines have been augmented with dummy vertices and have their source and sinks synchronized with each other. Then the $k+1$-composed machine can be defined to be $M^{k+1*} = \langle V^{k+1*}, U^{k+1*}, E^{k+1*}, s^{k+1*}, t^{k+1*}, A^{k+1*}, B^{k+1*} \rangle$, where

- $V^{k+1*} \subseteq V^{k*} \times V^{i+k}$.
- $U^{k+1*} \subseteq (U^{k*} \times U^{i+k}) \cup (U^{k*} \times V^{i+k}) \cup (V^{k*} \times U^{i+k})$.
- E^{k+1*} is the set of edges.
- $s^{k+1*} = \langle s^{k*}, s^{i+k} \rangle$.
- $t^{k+1*} = \langle t^{k*}, t^{i+k} \rangle$.
- $A^{k+1*} \subseteq \{ \langle w^{k*}, \beta^{i+k}(s^{i+k}) \rangle \mid \langle w^{k*}, \beta^{i+k}(s^{i+k}) \rangle \in U^{k+1*} \}$.
- $B^{k+1*} = \{ \langle b^{k*}, w^{i+k} \rangle \mid \langle b^{k*}, w^{i+k} \rangle \in U^{k+1*}$ and $b^{k*} \in B^{k*} \}$.

We will keep on composing these machines until the A-cutset moves beyond the B-cutset. The A-cutset is said to have moved beyond the B-cutset if: **1.** All paths from any event vertex in the B-cutset leads to an event vertex in the A-cutset and **2.** There does not exist an instantaneous path from any event vertex in the B-cutset to an event vertex in the A-cutset. In other words, the A-cutset is at least a state vertex away and to the right of the B-cutset. Let us assume that this happens after d compositions for the machine M^{d+1*}, as shown in Figure 7.5. For a serial pipeline this should imply that the system has d pipeline stages. Otherwise, the user gave an incorrect main machine.

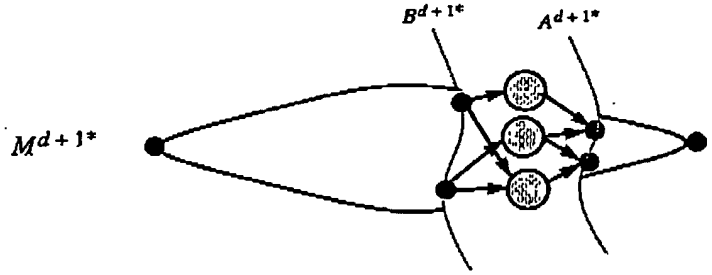


Figure 7.5: The resultant composed machine.

Consider our example of the simple processor with the fetch, decode, and execute pipeline stages shown earlier in Figure 7.2. The result of composing 4 of these main machines gives us the resultant composed machine shown in Figure 7.6. Notice that the cutset A^{4*} has moved beyond B^{4*}, and the state vertex $E^{i+1}D^{i+2}F^{i+3}$ is separating the two cutsets.

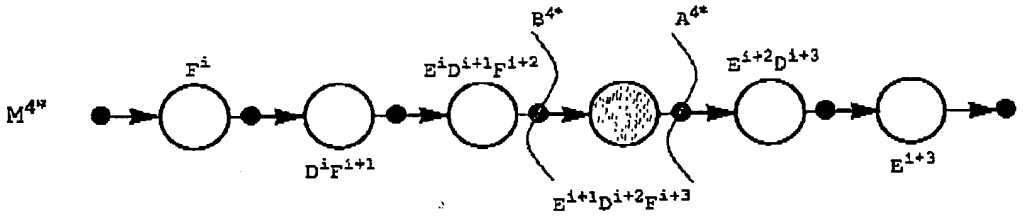


Figure 7.6: Example 1. The 4-composed machine for simple processor.

7.3.2 Normalization

The state vertices in the machine M^{d+1*} can be divided into 3 separate subsets, V_L, V_M, and V_R. The set V_L refers to the set of state vertices to the left of the B-cutset. The set V_M refers to the set of state vertices between the B and A-cutsets. And the set V_R are the set of vertices to the right of the A-cutset. We know that $V_L \subseteq V^i \times V^{i+1} \times \ldots \times V^{i+d-1}$ and $V_M \subseteq V^{i+1} \times V^{i+2} \times \ldots \times V^{i+d}$. Normalization consists of taking vertices in the set V_M and replacing them by incrementing edges to corresponding vertices in the set V_L. The shaded vertices in Figure 7.5 and Figure 7.6, just beyond the B-cutset, will be candidates for normalization. Consider an edge (u_b, v_m) where $u_b \in B^{d+1*}$ and $v_m \in V_M$. The state vertex v_m is of the form $\langle v^{i+1}, v^{i+2}, \ldots, v^{i+d} \rangle$. The vertex can be normalized to $\langle v^i, v^{i+1}, \ldots, v^{i+d-1} \rangle$. The normalized vertex should belong to the set V_L. And now we can replace the edge (u_b, v_m) by an incrementing edge from u_b to the normalized vertex. This construction gives us the closed form of the machine M^*.

Consider the example of the simple three-stage pipeline. The shaded vertex $E^{i+1}D^{i+2}F^{i+3}$ can be normalized to $E^iD^{i+1}F^{i+2}$ and we can introduce an incrementing edge as shown in Figure 7.7. Notice that this representation succinctly captures the system level view of the pipeline. Also, the representation captures all possible execution sequences.

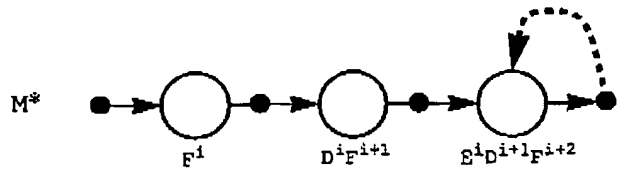


Figure 7.7: Example 1. System-level view of the pipeline for simple processor.

7.4 Examples

Consider the example of a processor which has two pipeline stages, namely, the fetch and execute stage and where an instruction might spend an arbitrary number of cycles in each stage. This example was introduced in Figure 3.12. The operation-level view of the main machine for such a processor is shown in Figure 7.8.

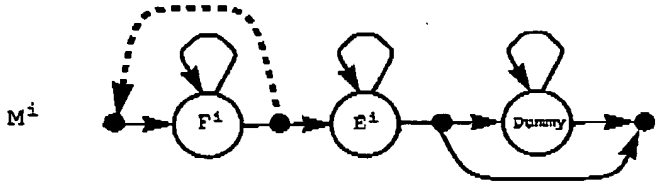


Figure 7.8: Example 2. Operation-level view of pipeline.

The closure construction for this machine gives us the machine shown in Figure 7.9.

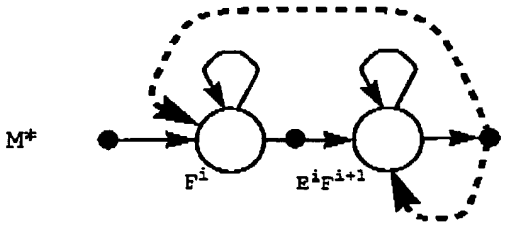


Figure 7.9: Example 2. System-level view of pipeline.

7.5 Summary

Closure construction was used to stitch main machines together to reason about execution sequences. This is only a partial solution or rather just the first step in the creation of execution sequences. Closure construction of the main machine just defines the effect of execution sequences on the pipeline structure. There is another aspect that requires checking whether constraints on the inputs and internal state elements are consistent in the creation of execution sequences. This requires some form of closure construction on the entire set of trajectory assertions. A more detailed discussion is presented as future work in Chapter 9.

THIS PAGE BLANK (USPTO)

Chapter 8

Cobra-Lite Verification

Our long-term objective is to use our methodology to verify the Cobra-Lite processor. The Cobra-Lite processor is implemented as a set of interacting functional units. A step towards our long-term objective is to verify individual functional units. This chapter applies our methodology to verify the fixed point unit in the Cobra-Lite processor. In particular, the chapter describes the verification of arithmetic and logical instructions with two source register operands and one target register operand in the fixed point unit.

The chapter starts with a brief overview of the Cobra-Lite processor with emphasis on the fixed point unit. The rest of the chapter describes our effort to verify the fixed point unit. The arithmetic and logical instructions were specified as a set of abstract assertions. Details of the instruction pipeline, forwarding logic, pipeline interlocks, and interface protocols were exposed in the implementation mapping. We, as the users, provided the abstract assertions and the implementation mapping[1]. The trajectory generation phase took the abstract assertions and implementation mapping and generated the trajectory assertions. Finally, Symbolic Trajectory Evaluation was used to verify the trajectory assertion on a gate-level circuit design of the fixed point unit. A brief description of our effort to verify the fixed point unit can be found in [23].

8.1 The Cobra-Lite Processor

Cobra-Lite is a fully functional prototype of the processor found in IBM's AS/400 Advanced 36 Computer[64][65]. It is a superscalar implementation of the PowerPC architecture[63]. The PowerPC architecture is a full 64-bit architecture with a well-defined 32-bit subset. The architecture has 32 general purpose registers, 32 floating point registers, 3 branch registers and 2 exception status registers. The architecture supports 5 different instruction formats and several addressing

1. Kyle Nelson, at IBM Rochester, is leading the effort to verify the Cobra-Lite processor.

modes. Cobra-Lite is an early version of the Cobra processor with the processor, cache, and I/O interface on a single chip. It is called Cobra-Lite since it is missing about 17 instructions from the required 64-bit PowerPC set. These instructions are primarily floating point instructions that were implemented in software.

The Cobra-Lite processor has many of the complicating features found in modern processors such as forwarding logic, instruction pipelines, pipeline interlocks, multiple cycle instructions, multiple instruction issue, conditionally issued instructions, and out-of-order execution. Cobra-Lite is implemented as a set of interconnected functional units. A high-level diagram of the dataflow between some of these functional units is shown in Figure 8.1.

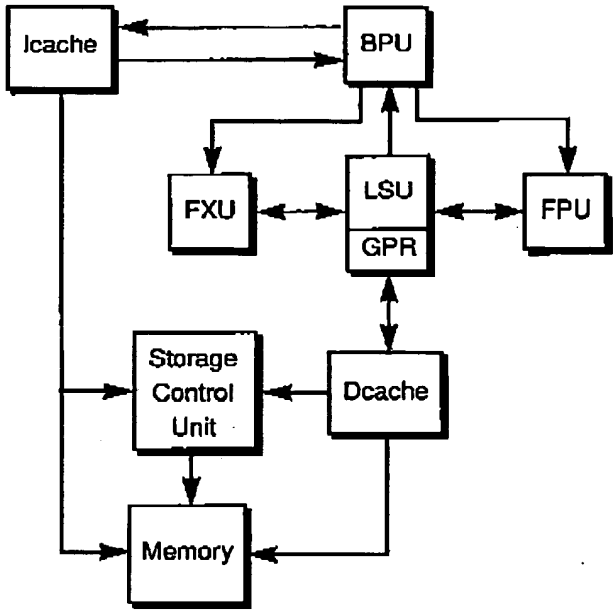


Figure 8.1: High-level dataflow between major functional units and caches.

The initial attempt will be to individually verify three of these functional units, i.e., the fixed point unit (FXU), load store unit (LSU), and branch processing unit (BPU). The floating point unit (FPU) is outside the scope of the languages and tools described in this thesis. Our Symbolic Trajectory Evaluator uses ordered Binary Decision Diagrams to perform the symbolic computation. Binary Decision Diagrams are not suited to represent floating point functions. Recently researchers have explored alternate data structures to represent floating point functions[75]. An attempt to

verify the FPU would require the incorporation of these data structures into Symbolic Trajectory Evaluation.

The implementation of the Cobra-Lite processor is described in the VHDL language that was synthesized down to the gate level. A measure of complexity of the processor is the number of gates in individual functional units. The FXU circuit has 22,942 multi-input gates and 1092 latches. The LSU circuit is even more complex and has 85,487 multi-input gates and 6912 latches.

The next few sections briefly describe the three functional units, i.e., the BPU, LSU, and FXU with emphasis on the FXU.

8.1.1 Branch Processing Unit

The branch processing unit (BPU) fetches up to two instructions per cycle from the Instruction Cache (Icache). The BPU can pre-fetch up to four sequential instructions or two branch target instructions. A maximum of two instructions can be dispatched per cycle depending on the number of valid pre-fetched instructions and interlock conditions set by previously dispatched instructions. All instructions are pre-decoded and steered to the correct functional unit. If a branch is detected in the instruction stream and the effective address is available, the BPU will attempt a conditional fetch of the target instruction. Instructions following the branch are dispatched conditionally. When the branch is resolved, instructions dispatched conditionally are allowed to execute if the branch is not taken, or are cancelled if the branch is taken. The BPU also implements a number of architected registers (e.g., condition code registers) and interfaces to the FXU, FPU, and LSU to allow access to these registers.

8.1.2 Load Store Unit

The purpose of the load store unit (LSU) is to execute the subset of the architected fixed point instructions that access main storage. These instructions are dispatched to the LSU by the BPU. The LSU decodes these instructions, performs the necessary effective address calculation, and interfaces with the Data Cache (Dcache) and the Storage Control Unit.

In the Cobra-Lite architecture all targets for loads and sources for stores are GPRs. There are thirty-two 64-bit GPRs in the architecture. These registers are contained in a sub-chiplet of the LSU. The LSU is responsible for the bookkeeping of the GPRs so that even though the processor is executing instructions simultaneously or possibly out of sequence, the GPRs contain the architecturally correct value should an interrupt occur. This is done by a scoreboarding scheme that marks GPRs as busy when an update is pending and frees them when the update is complete.

The FXU needs to access the GPRs to perform arithmetic and logical operations. To provide this, the LSU has an interface with the FXU that allows the FXU to request GPRs for operand sources and store back GPRs for destinations. The LSU is responsible to check if the GPR is busy prior to supplying it to the FXU and marking the destination GPR busy while the FXU is performing the operation.

The LSU circuit has in excess of 80,000 multiple input gates and nearly 7000 latches.

8.1.3 Fixed Point Unit

The fixed point unit (FXU) is responsible for executing all fixed point instructions other than loads and stores. The FXU interfaces with the BPU and LSU. Instructions are received from the BPU with an acknowledge handshake. If the FXU is busy and cannot receive the instruction, then no acknowledgment will be given to the BPU. The instruction will stall in the dispatch stage, and the BPU will try to dispatch it again in the following cycle.

The FXU processes instructions in three stages, i.e., the dispatch, decode, and execute stages. The instruction is received from the BPU in the dispatch stage. In the decode stage, the latched instruction is decoded into the instruction fields. Requests are made to the LSU, where the general purpose registers (GPRs) reside. A target operand, if required, is also reserved in this stage. The required register source operands will come from one of two places. If the source register address is equivalent to the target register address of the instruction in the execute stage, then the source data will be forwarded to the decode stage, bypassing the register file. Otherwise, if there is no register match or if there is some special circumstance in which the hardware does not support register bypassing, a request for the data is sent to the LSU. Part of the LSU's function is to manage all

the interlocks involved in the allocation of register resources. Instructions will stall in the decode stage until all the source operands are available and the execute stage is available. The execute stage is considered available when either it does not currently contain a valid instruction or the instruction in the execute stage is going to be completed in the current cycle.

When the execute stage begins, all required operands are available and the current instruction is latched into the execute stage instruction register. Decoded fields from the decode stage are latched to prevent having to re-decode them. Most instructions will complete the execute stage in a single cycle. Multiply and divide instructions are the exceptions. Result data is provided to the LSU and the instruction is completed when the LSU acknowledges the store. This acknowledgment may be delayed, causing the instruction to stall, if there are previous instructions in the instruction stream that still have the possibility of causing an interrupt.

The FXU can execute one instruction per cycle if instructions are available, no contention occurs for GPR resources, and there are no delays due to interrupt possible conditions. The FXU can complete instructions ahead of the LSU until a resource contention condition is encountered.

The FXU circuit has in excess of 20,000 multi-input gates and in excess of 1000 latches.

The complexity resulting from both the interlock logic that enforces pipeline stalls and the nondeterministic interfaces between the FXU and other functional units are common in modern processors. These features are a significant source of errors and increase the difficulty of the verification task.

8.2 Abstract Specification

This chapter concentrates on verifying three-register instructions in the fixed point unit. Three-register instructions are arithmetic and logical instructions with two source register operands and one target register operand. Each three-register instruction is specified as an abstract assertion. As an example, the abstract assertion for the three-register bitwise-OR instruction is as follows:

```
WHEN ((ra != rb) || (dataA == dataB))                    (1)
  (op is OR) and                                         (2)
   (RA is ra) and (RB is rb) and (RT is rt) and           (3)
   (Reg[ra] is dataA) and (Reg[rb] is dataB)             (4)
     LEADSTO                                             (5)
  (Reg[rt] is dataA | dataB)                             (6)
```

Lines 2-4 contain the precondition of the abstract assertion. Line 2 specifies that the opcode must be that for the OR instruction. Line 3 uses the symbolic variables, `ra`, `rb`, and `rt`, to specify that the source operands, RA and RB, and the target operand, RT, can be any register address. Line 4 specifies the contents of the two source registers to be the symbolic values `dataA` and `dataB`. Line 6 specifies that in the postcondition the content of the target register will contain the bitwise-OR of the data in the two source registers. Line 1 is a global domain restriction that filters out illegal patterns. An illegal pattern would be when the two source operands refer to the same register address and the data contained in the two source registers is different.

In the remainder of this chapter, we will refer to the three-register instruction being verified as the Instruction Under Test (IUT). The three-register bitwise-OR instruction will serve as a representative IUT.

8.3 Implementation Mapping

Figure 8.2 shows the high-level flow of information between the BPU, LSU, and FXU for the instruction under test (IUT). The FXU has three pipeline stages, i.e., the dispatch, decode, and execute stages. The FXU receives the instruction from the BPU in the dispatch stage. The instruction has the opcode and the three-register addresses. The FXU obtains the source operands from the LSU in the decode stage. The operation is performed in the execute stage and the result is stored back in the target register.

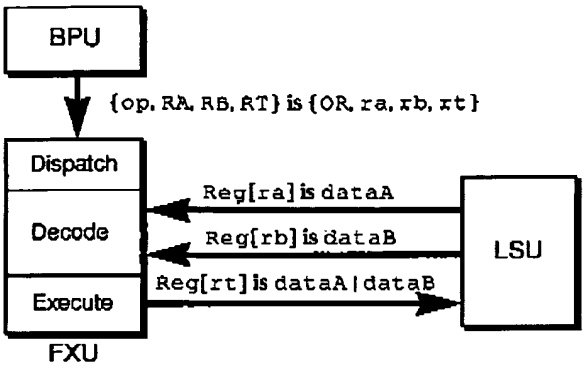


Figure 8.2: High-level information flow for instruction under test.

The implementation mapping has to relate the high-level information flow to a transfer of logic values on actual signals in the circuit. The signals involved in these transactions are shown in Figure 8.3. The abstract clauses are mapped into protocols involving the interface signals between the FXU, BPU, and LSU. The abstract clause `{op, RA, RB, RT} is {OR, ra, rb, rt}` is mapped into a protocol on interface signals between the BPU and FXU. The abstract clauses `(Reg[ra] is dataA)`, `(Reg[rb] is dataB)`, and `(Reg[rt] is dataA|dataB)` are mapped into protocols on interface signals between the FXU and LSU. A detailed description of each signal will be given later while describing individual map machines.

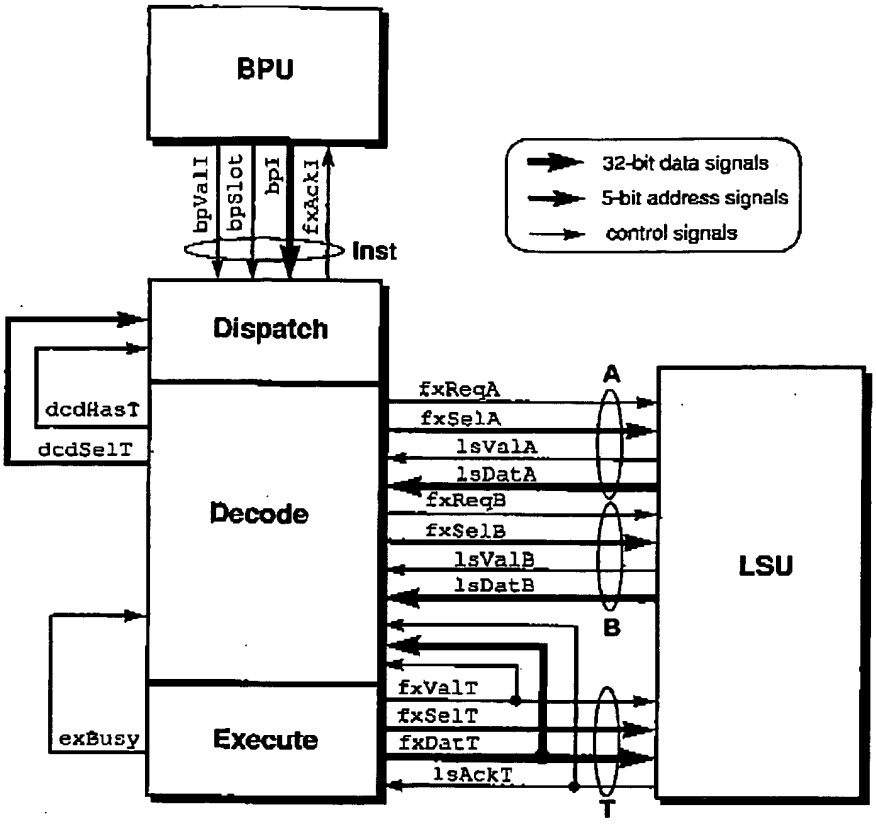


Figure 8.3: The set of signals exposed in the implementation mapping.

Our intention is to verify the IUT under every possible sequence of leading instructions. One possible way to represent all leading instruction streams, is to issue two completely symbolic instructions before issuing the IUT to the FXU. The problem with this approach is that the symbolic computation required for the leading instructions is prohibitively large. Therefore, we capture all possible leading instructions streams by exposing and asserting some of the internal state elements in the FXU. Note that only a limited number of internal state elements interact with the IUT. Apart from the interface signals, we had to expose only 8 internal state elements in the FXU. Some of the more important internal state elements that have to be exposed in the mapping are shown in Figure 8.3. The internal signals `dcdHasT` and `dcdSelT` are asserted in the dispatch stage and checked in the decode stage. The internal signal `exBusy` is asserted in the decode stage and checked in the execute stage. Again a detailed description of these signals will be presented later while describing individual map machines.

The next few sections describe the details of the implementation mapping for the FXU. The mapping describes a cycle-level state vertex. The main machine and set of map machines are defined in terms of cycle-level state vertices. A map machine is defined for each abstract element in the abstract specification. In addition, map machines are also defined for some of the internal state elements of the FXU that are exposed in order to capture the past history of the processor.

8.3.1 Cycle-Level State Vertex

The Cobra-Lite processor has a two-phase latch structure. Each storage element is two latches in a master-slave connection. The master latches are called the L1 latches and the slave latches are called the L2 latches. The processor uses two clock signals, i.e., the CL1 and CL2 signals, to operate these latches. A cycle-level vertex is defined in terms of two phase-level vertices, L1 and L2, as shown in Figure 8.4.

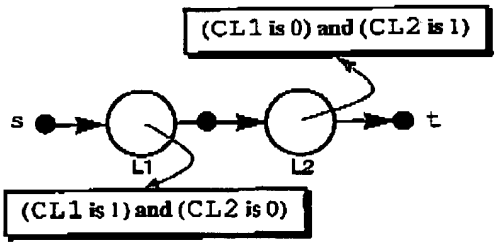


Figure 8.4: Cycle-level vertex for the FXU.

8.3.2 Main Machine

The main machine for the FXU is shown in Figure 8.5. The vertices dsp, dcd, and exe are cycle-level state vertices which represent the three pipeline stages in the FXU. The self loops on each state vertex represent that an instruction can stall in each pipe stage for a nondeterministic number of cycles. The successive instruction can be started at event vertex m2, thus overlapping the decode stage of the current instruction with the dispatch stage of the successive instruction.

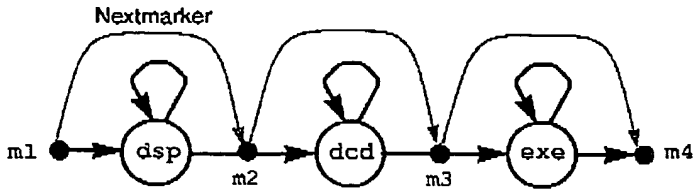
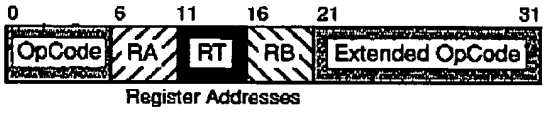


Figure 8.5: Main machine for the FXU.

8.3.3 Dispatch Stage Map Machines

The abstract clause (`op is OR`) is mapped into a protocol on the interface signals between the BPU and FXU. The signals involved in this transaction are shown in Figure 8.3. The BPU asserts the signal `bpValI` when an instruction has to be dispatched to the FXU. The signal `bpValI` is kept asserted until the FXU acknowledges receiving the instruction by asserting the signal `fxAckI`. In addition to `bpValI`, the BPU also asserts the `bpSlot` signal. The BPU can dispatch two instructions per cycle. The signal `bpSlot` indicates the slot for the dispatched instruction. The 32-bit `bpI` signal defines the opcode and operands for the instruction. The format for three-register instructions is shown below:



This transaction is captured in the map machine *Map*(`op is opcode`) . A textual description of a part of the map machine is given in Figure 8.6. The map machine defines a control graph in terms of cycle-level state vertices. The control graph is synchronized with the dispatch stage of the main machine. The labels define the action and reaction node formulas associated with the state vertices. The node formulas are dependent upon the opcode of the instruction.

```
MAP (op IS opcode) {
    VERTEX S1, S2, S3: cycle;
    VERTEX e1, e2, e3, e4: EVENT;
    NEXT {
        e1: {S1, e2};
        S1: {S1, e2};
        e2: {S2, e3};
        S2: {S2, e3};
        e3: S3;
        S3: e4;
    }
    SYNCH {e1: main.m1, e4: main.m2};
    VARIABLE slt: BIT;
    CASE (opcode) {
        IS OR:
            ALABEL {
                (bpValI is 0 at S1.L1) and (bpValI is 1 at S2.L1) and
                (bpValI is 1 at S3.L1) and (bpSlot is slt at S3.L1) and
                (bpI[0:5] is "011111" at S3.L1) and (bpI[21:31] is "01101111000" at S3.L1)
            }
            CLABEL {
                (fxAckI is 0 at S1.L1) and (fxAckI is 0 at S2.L1) and (fxAckI is 1 at S3.L1)
            }
        IS AND:
            ...
        ...
    }
}
```

Figure 8.6: Map machine for (op is opcode).

Map machines for the register addresses, RT, RA, RB can be described in a similar manner. All these map machines are synchronized to the dispatch stage of the main machine. The map machines define the trajectory formula for each simple abstract formula in the abstract assertion. The trajectory formulas for the operation and the register addresses is shown pictorially in Figure 8.7. The next few paragraphs discuss each of these trajectory formulas.

The trajectory formula $\mathcal{TF}$(op is OR) is derived from the map machine $\mathcal{M}ap$(op is opcode). The control graph has three state vertices, S1, S2 and S3. The action node formulas associated with each state vertex are shown in the upper half and the reaction node formulas are shown in the lower half of the shaded box. The action node formula on state vertex S2 asserts the signal bpValI. The action node formula on state vertex S3 sets the opcode and extended opcode for the three-register bitwise-OR instruction. The signal bpSlot is asserted to a symbolic variable slt to indicate that instruction might have been dispatched in either slot. The BPU might have to wait for an indefinite number of cycles before the FXU acknowledges the instruction. The check on the

signal fxAckI in the reaction node formula on state vertex S3 ensures that the instruction is eventually acknowledged.

The trajectory formula $\mathcal{TF}$(RT is rt) sets appropriate fields in the bpI signal for the target register address.

The decision to fetch the source operands from the register file or forward the data is made in the dispatch stage. Forwarding is dependent upon the previous instruction. The two instructions involved in the decision are the previous instruction in the decode stage and the IUT in the dispatch stage. Eventually the previous instruction will move into the execute stage and forward the data to the IUT in the decode stage. The FXU requires the following conditions to forward data from the execute stage:

1. Previous instruction should have a target register address. The internal signal dcdHasT indicates whether the previous instruction had a target register.
2. A proper sequencing of instructions. In particular, two FXU instructions have to be issued in consecutive cycles by the BPU with no intervening LSU instruction.
3. The target register address of the previous instruction should match the source register address of the IUT. The internal signal dcdSelT stores the target register address of the previous instruction.

The clause (PrevI is Any) is appended to the precondition to denote an arbitrary previous instruction. The equivalent clause appended to the postcondition is (PrevI is OR) since we know that the previous instruction in the postcondition is the IUT, i.e. the bitwise-OR instruction. The map machine for the abstract element PrevI exposes the internal signals dcdHasT and dcdSelT in the FXU. The map machine is used to derive trajectory formulas for both the precondition and postcondition. The trajectory formula $\mathcal{TF}$(prevI is Any) sets the internal signals to indicate an arbitrary previous instruction. The action node formula on state vertex S1 sets the signals dcdSelT and dcdHasT to symbolic variables prt and trg respectively. The trajectory formula $[\mathcal{TF}(\texttt{prevI is OR})]^M$ has a reaction node formula to check the internal signals. The signal dcdSelT is required to be the target register address of the OR instruction, i.e., the sym-

bolic variable rt. The signal dcdHasT is required to be 1 to indicate that the OR instruction has a target register address.

The trajectory formulas $\mathcal{TF}$(RA is ra) and $\mathcal{TF}$(RB is rb) sets the appropriate fields in the bpI signal for the A and B source register addresses. The trajectory formulas have two legs. The top leg, with state vertices S1 and S2, represents the case where the source operand will be fetched from the LSU. The bottom leg, with state vertex S3, represents the case where the source operand will be forwarded from the execute stage. The state vertex S3 sets the source address to the symbolic variable prt, i.e., the target address of the previous instruction. Since the FXU requires two instructions to be received in consecutive cycles to permit bypassing, the bottom leg does not permit any stalling in the dispatch stage.

This section has presented a somewhat simplified view of bypassing in the FXU. The actual details are a bit more complicated. A few more signals are required to ensure that there are no intervening LSU instructions between the two FXU instructions.

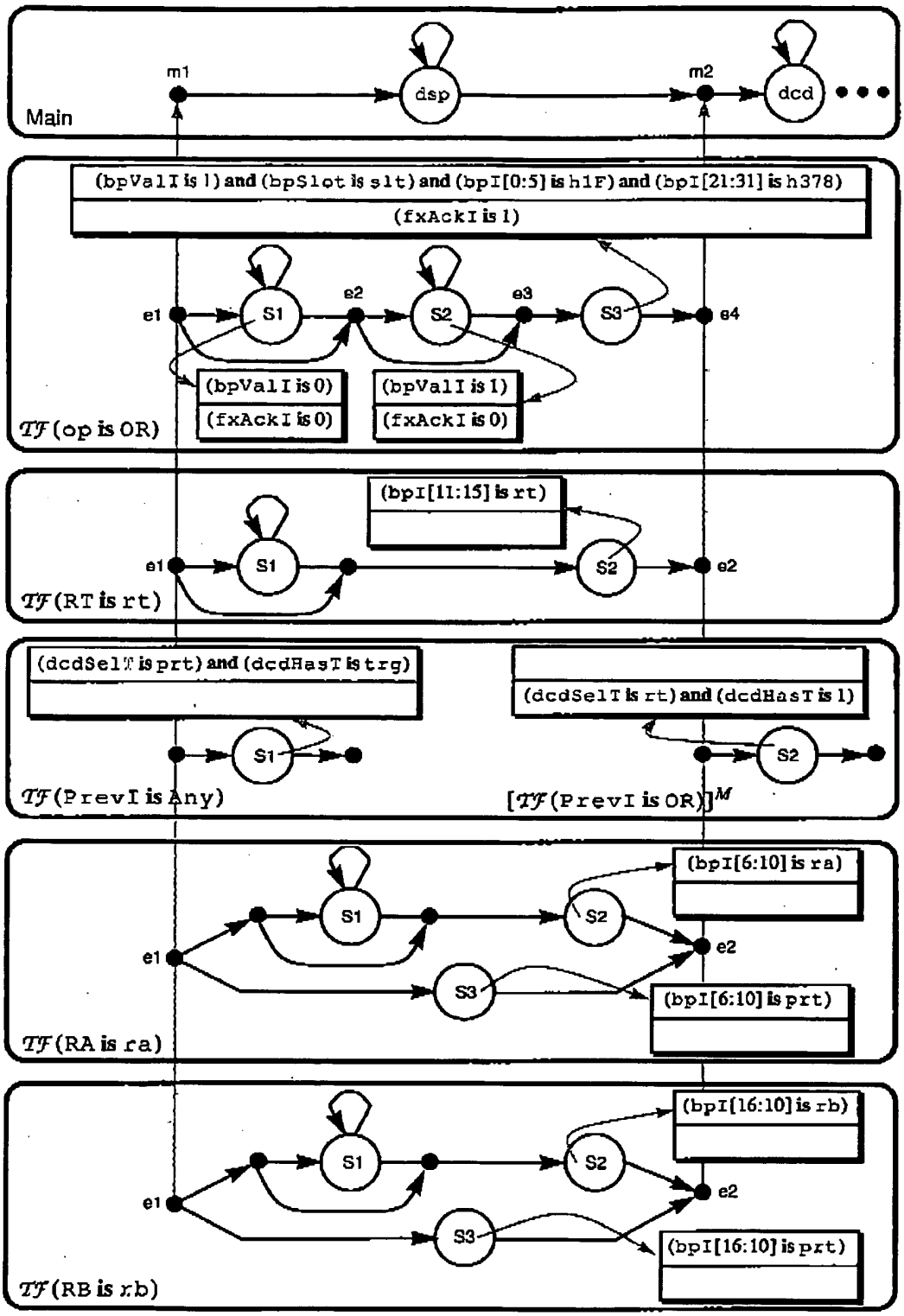


Figure 8.7: Dispatch stage map machines.

8.3.4 Decode Stage Map Machines

The abstract clauses (Reg[ra] is dataA) and (Reg[rb] is dataB) are mapped into protocols on the interface signals between the FXU and LSU. The signals involved in this transaction are shown in Figure 8.3. The data operands can either be obtained from the LSU or forwarded from the execute stage. Consider the A operand data. If the data has to be obtained from the LSU, then the FXU will assert the signal fxReqA and set the signal fxSelA to the register address. When the data is available, the LSU will assert the signal lsValA and send the data on the signal lsDatA. If the data has to be forwarded, the execute stage will assert the signal fxValT to indicate that the signal fxDatT has the required data operand.

The trajectory formulas for the data operands is shown in Figure 8.8. The trajectory formulas are synchronized to the decode stage of the main machine. The action node formulas are shown in the upper half of the shaded box. The reaction node formulas are shown in the lower half of the shaded box. Consider the trajectory formula for the A operand, $\mathcal{TF}$(Reg[ra] is dataA) . The trajectory formula has two legs. The top leg, with state vertices S1 and S2, represents the case where the A operand is obtained from the LSU. The reaction node formulas on state vertices S1 and S2 check that the FXU makes a request for the A operand from the symbolic address ra in the register file. The action node formulas on state vertices S1 and S2 indicate that the LSU waits for an arbitrary number of cycles before indicating that a valid data has been placed on the signal lsDatA. The bottom leg, with state vertices S3 and S4, represents the case where the A operand is forwarded from the execute stage. The execute stage waits for an arbitrary number of cycles before placing the forwarded data on the signal fxDatT. The two legs converge into the state vertex S5. State vertex S5 represents the condition where the decode stage has obtained the A operand but might still be waiting to obtain other resources such as the B operand. The trajectory formula $\mathcal{TF}$(Reg[rb] is dataB) can be explained in a similar fashion.

The IUT can move from the decode stage to execute stage if the execute stage is not busy. However, if the execute stage is busy, then the IUT will stall in the decode stage. If the execute stage is busy, the IUT cannot move to the execute stage until the decode stage receives the fxValT and lsAckT signals. The fxValT signal indicates that the execute stage has generated the result. The lsAckT signal indicates that LSU has accepted and stored the result in the register file. This con-

dition is represented in the trajectory formula $\mathcal{TF}$(PrevI is Any) for the execute stage. The trajectory formula has two legs. The bottom leg, with state vertices S1, S2, and S3, represents the case where the execute stage is busy. A valid target is generated in state vertex S2. The LSU sends the acknowledgment in state vertex S3. The top leg represents the case where the execute stage is not busy, i.e., did not have a valid instruction. The two legs converge into state vertex S4. The state vertex S4 represents the case where the execute stage is available, but the IUT might be waiting for other resources such as the source operands.

The above discussion has ignored some of the complications arising due to interactions between data forwarding and the signals exBusy and fxValT. Assume that the A operand is being forwarded from the execute stage. This implies that we are on the bottom leg of the trajectory formula $\mathcal{TF}$(Reg[ra] is dataA) . Forwarding requires a valid instruction in the execute stage. This requires us to restrict the composition of the bottom leg of the trajectory formula $\mathcal{TF}$(Reg[ra] is dataA) to the bottom leg of the trajectory formula $\mathcal{TF}$(PrevI is Any). Also, forwarding cannot occur until the execute stage has a valid data, i.e., the signal fxValT has been asserted. This requirement can be met by synchronizing event vertex e2 in the trajectory formula $\mathcal{TF}$(Reg[ra] is dataA) with event vertex e2 in the trajectory formula $\mathcal{TF}$(PrevI is Any) . However, remember that the synchronization function is from event vertices in the map machine to event vertices in the main machine. We end up complicating the main machine by dividing the decode stage into two legs. One leg represents the case where the source operand is being forwarded whereas the other leg represents the case where the source operand is obtained from the LSU. And the bottom legs of the map machines are synchronized to the forwarding leg of the main machine.

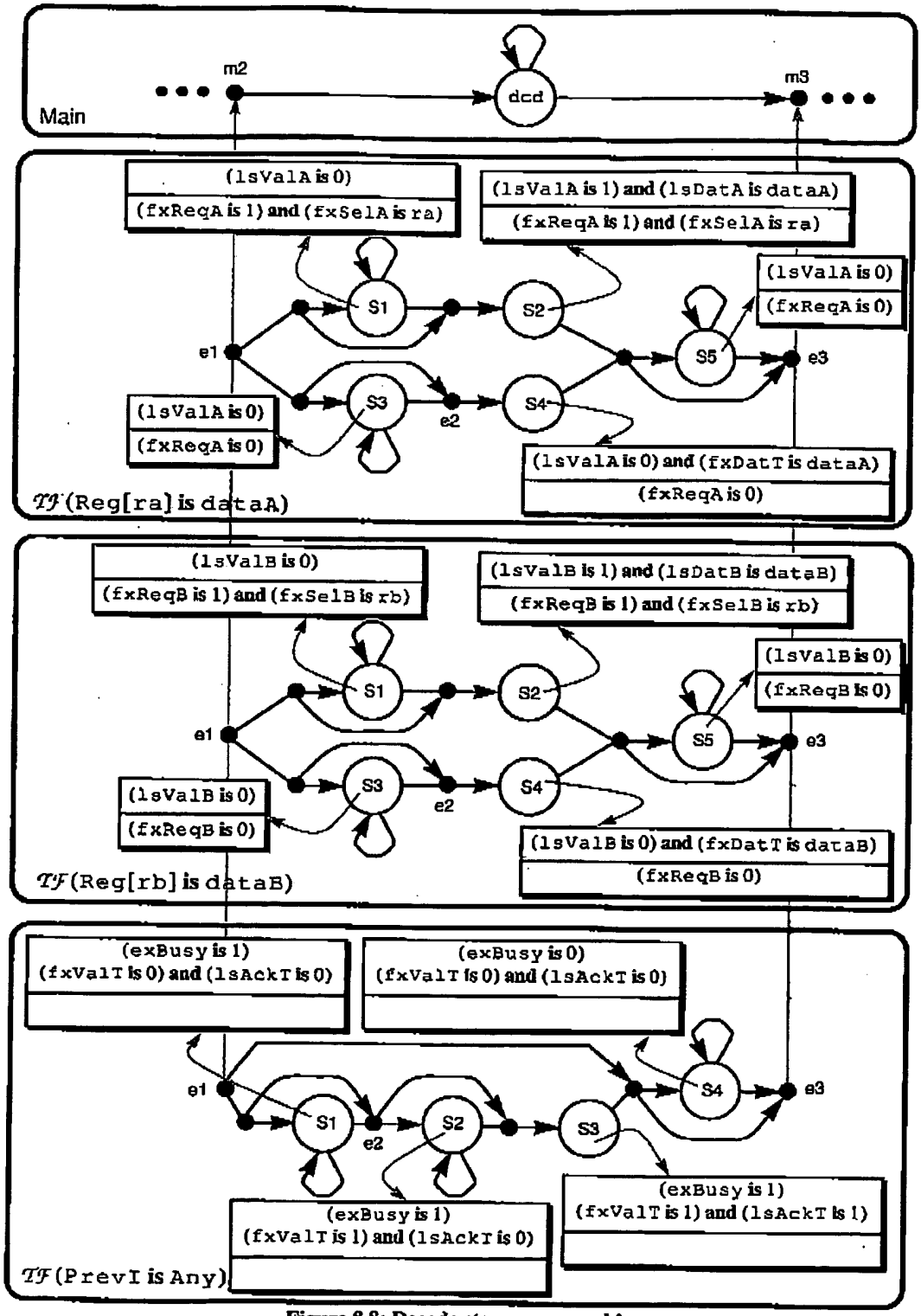


Figure 8.8: Decode stage map machines.

8.3.5 Execute Stage Map Machines.

The postcondition clause `(Reg[rt] is dataA|dataB)` is mapped into a protocol between the FXU and LSU. The signals involved in this transaction are shown in Figure 8.3. The execute stage indicates that the target data has been generated by asserting the signal `fxValT`. The register address is placed on the signal `fxSelT` and the target data is placed on the signal `fxDatT`. The LSU indicates that the data has been stored by asserting the `lsAckT` signal.

The trajectory formula for the postcondition is shown in Figure 8.9. The reaction node formulas on state vertices `S1` and `S2` check that the FXU has generated the result of the bitwise-OR operation and is asking the LSU to store the result in the target address in the register file. The LSU might wait for an arbitrary number of cycles before storing the result. The action node formula on state vertex `S2` acknowledges the end of the transaction.

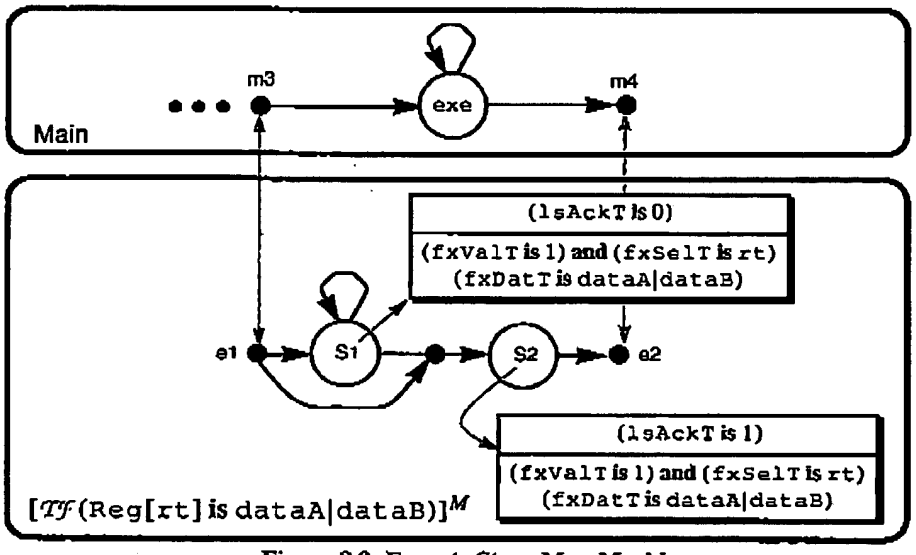


Figure 8.9: Execute Stage Map Machines.

For ease of exposition, the above description of the implementation mapping has been simplified to some extent. Some of the more subtle signals that deal with forwarding, multi-cycle instructions and effects of stalling in the LSU have been ignored in this description. The complete implementation mapping for three-register instructions has 24 map machines. The textual description consists of approximately 600 lines of map code.

8.4 Trajectory Generation

The trajectory generation phase took the abstract assertions and the implementation mapping and generated the trajectory assertions. The trajectory assertion, shown in Figure 8.10, corresponds to all possible cases that can arise due to interactions between the map machines. The trajectory assertion has 224 cycle-level vertices and 34 loops. The corresponding acyclic component graph has 28,602 paths from source to sink. Each path represents a unique ordering of events for a particular set of inputs and constraints on internal state elements.

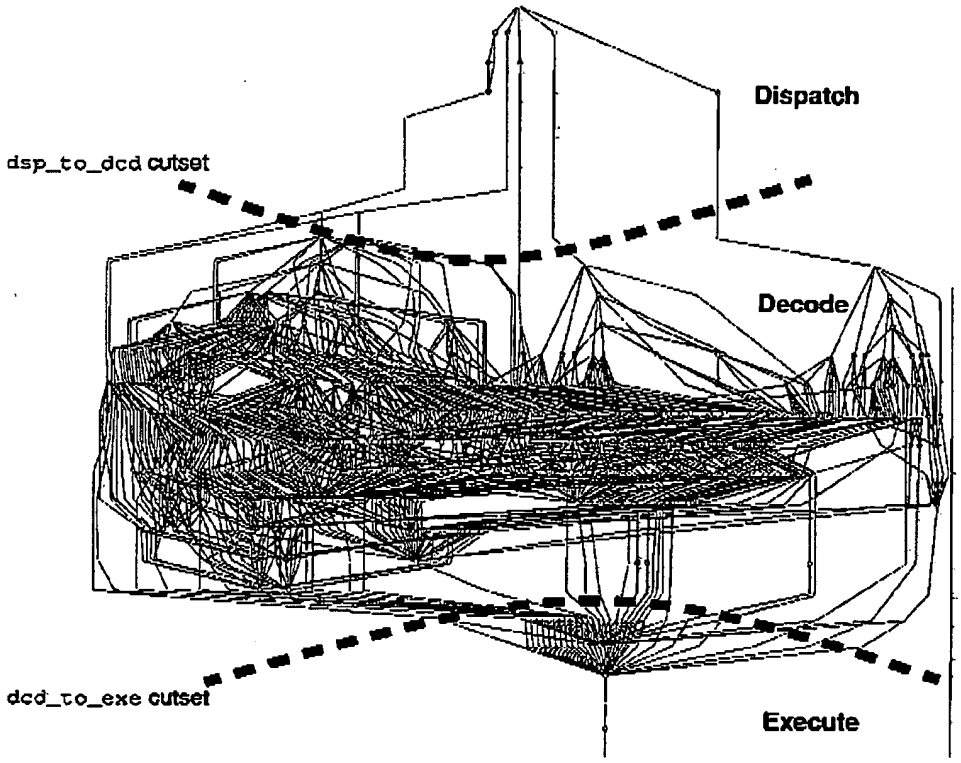


Figure 8.10: The trajectory assertion for three-register instructions.

Some intuitive sense can be made out of the graph. Notice the horizontal cutsets in the graph. The `dsp_to_dcd` cutset corresponds to the event vertex `m2` in the main machine. The `dcd_to_exe` cutset corresponds to the event vertex `m3` in the main machine. The cutsets divide the trajectory assertion into the dispatch, decode, and execute stages. The decode stage of the graph is significantly more complex than the dispatch and execute stages. The reason is that the bulk of the control logic in the circuit resides in the decode pipeline stage. The decode pipeline stage is

responsible for obtaining and reserving multiple resources. The various possible orderings in which these resources can be obtained and reserved accounts for the complexity of the decode stage. The dispatch and execute stages, on the other hand, require a single acknowledgment in order to complete the pipeline stage.

A high-level view of the trajectory assertion is shown in Figure 8.11. The figure shows the details of the dispatch and execute stages. The decode stage is represented by the shaded block.

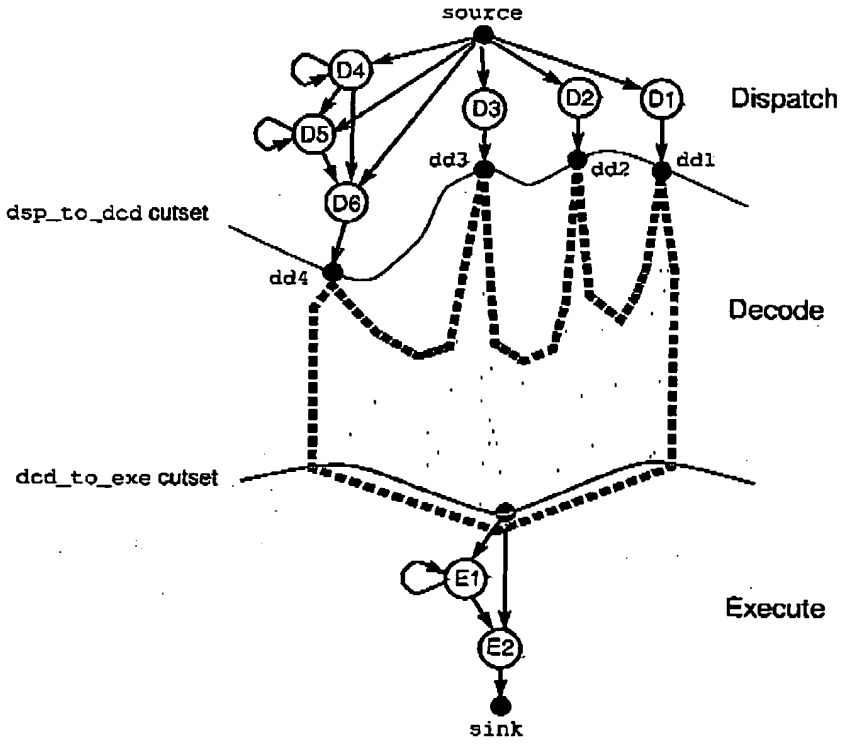


Figure 8.11: High-level view of the trajectory assertion for three-register instructions.

The dispatch stage can be divided into 4 vertical slices. Slice 1 is the rightmost slice from the source to event vertex dd1. Slice 2 is from the source to event vertex dd2. Slice 3 is the one from the source to event vertex dd3. And slice 4 is the leftmost slice that has all paths from the source to event vertex dd4. Slices 1, 2, and 3 are simple slices with only a single state vertex. These simple slices represent the case where at least one of the source operands are going to be forwarded. The FXU requires two instructions to be received in consecutive cycles for forwarding to occur. The state vertices D1, D2, and D3 represent the case where the FXU received and acknowledged

the instruction in a single cycle. Slice 4 represents the case where both operands are being fetched from the LSU. The state vertices D4, D5 and D6 capture the protocol between the FXU and BPU. The FXU receives the instruction in state vertex D5. The FXU can wait for an arbitrary number of cycles before acknowledging the instruction in state vertex D6.

The 4 slices in the dispatch stage define 4 entry points into the decode stage. Let Group 1 refer to the set of paths in the graph between event vertex dd1 and the sink. Similarly, let Group 2, Group 3, and Group 4 refer to the set of paths in the graph from event vertices dd2, dd3, and dd4 to the sink. Group 1 corresponds to the case where the A source operand is being forwarded while the B source operand is fetched from the LSU. Group 2 corresponds to the case where the B source operand is being forwarded while the A source operand is fetched from the LSU. Group 3 corresponds to the case where both source operands are forwarded. And Group 4 corresponds to the case where both operands are fetched from the LSU. Group 4 is the most complex of these groups. This is because all the resources that are needed in the decode stage are completely independent. The paths have to capture all possible arrival orders and timings for each and every resource. Group 3 is the simplest of these groups. In this group, the arrival of the A and B source operands coincides with the signal fxValT generated by the execute stage.

The execute stage has two state vertices E1 and E2. The result of the bitwise-OR operation is available in these state vertices. The FXU has to wait until the LSU acknowledges the target in state vertex E2.

8.5 Symbolic Trajectory Evaluation

The trajectory assertion in Figure 8.10 is very complex and has in excess of 28000 paths in the corresponding acyclic component graph. It would be computationally infeasible to enumerate all the paths and use STE separately on each path. Instead, STE encoded the graph using 456 path variables. STE was used to verify the entire trajectory assertion in a single verification run. The loops in the trajectory assertion were dealt by performing a greatest fixed point computation. The verification of the three-register bitwise-OR instruction took nearly 50 hours of CPU time and 185 MBytes of memory on an IBM RS/600 43P Model 140.

This may seem to be a considerable amount of time and memory. However, note that the bitwise-OR operation was verified under all possible bypass, interlock, and pipeline conditions and under all possible timings for interface with the BPU and LSU. Both the data path and control were verified using BDDs. The CPU time appears to be more reasonable when we consider the enormous number of paths that are being verified. On an average, STE is able to verify a path in 6-7 seconds. An additional point is that verification of the complete trajectory assertion would most likely be run at the end as a regression test. During the development and debugging phase, STE can be run interactively to debug parts of the design by focussing the verification on problematic paths.

Our Symbolic Trajectory Evaluator is a first generation tool that was developed more from the point of view of completeness rather than efficiency. Several techniques could be used to improve the efficiency and effectiveness of STE: **1.** It seems possible to develop a levelized version of STE that would considerable reduce the number of next-state computations. **2.** Our version of STE uses a simple-minded user-defined variable ordering. Advanced variable ordering heuristics and dynamic variable reordering could be used to reduce the complexity of the symbolic computations. **3.** Various forms symmetry and abstraction transformations could be used to simplify the formal verification task. Details of all these techniques are discussed as future work in Chapter 9.

As an aside, it is not necessary to have a one-to-one relationship between instructions and assertions. A single assertion could be used to specify a class of instructions with the same instruction format. This would significantly reduce the number of assertions to be verified with a slight increase in the CPU time.

8.6 Summary

This chapter applied our methodology to verify arithmetic and logical instructions in the fixed point unit of the Cobra-Lite processor. The specification was kept abstract at the level of the instruction set architecture. The mapping provided the complex implementation specific details for the FXU. STE was used to verify that the FXU circuit correctly fulfilled the abstract specification of the processor. In some sense, the mapping merely served as hints to guide the verification task.

At first glance, the implementation mapping might seem to be too complex. However, note the fact that most of this complexity is in defining the environment around the FXU. The reality is that modern processors are designed as a set of reactive subsystems with complex interfaces and protocols. Therefore, any technique for formal verification of subsystems would have to deal with the same level of complexity.

A large part of the complexity in the trajectory assertion is due to pipeline stalls. The trajectory assertion represents all possible permutations in which various different resources can be obtained to resolve pipeline stalls. Our approach seems to be the first one that can truly deal with the complexity of pipeline interlocks.

THIS PAGE BLANK (USPTO)

Chapter 9

Conclusions and Future Work

9.1 Conclusions

This thesis has presented a methodology for formal hardware verification using Symbolic Trajectory Evaluation. The methodology is targeted towards systems that have a simple deterministic high-level specification but have implementations that exhibit highly nondeterministic behaviors. There are three main aspects of our methodology: 1. Specification Languages 2. Tools 3. Theory.

The specification is divided into two components, i.e., the abstract specification and the implementation mapping. The abstract specification defines the high-level behavior of the system, which is specified as a set of abstract assertions in a Hardware Specification Language. Each abstract assertion defines the effect of an operation on the user-visible state. Implementation specific details such as the system clocking, pipeline structure, and interface protocols are captured in the implementation mapping. The implementation mapping provides a temporal and spatial mapping for abstract elements in the high-level specification. The mapping is defined in terms of control graphs, which are state diagrams with the capability of synchronization at specific time points.

The abstract specification and the implementation mapping are provided by the user. Once the descriptions have been given, the tools take over, i.e., the trajectory generator and the Symbolic Trajectory Evaluator. The trajectory generator takes in the abstract specification and implementation and generates the trajectory specification, which consists of a set of trajectory assertions. Each abstract assertion gets mapped into a trajectory assertion. The trajectory assertion captures all possible nondeterministic interactions that can arise in the implementation. The Symbolic Trajectory Evaluator is used to verify each individual trajectory assertion on the actual circuit design. Symbolic Trajectory Evaluation (STE) can be considered to be a hybrid approach, which combines symbolic simulation with some of the capabilities of model checking. The main advantage of STE is that it avoids the need to build the next-state function for the entire circuit.

Once the tools have verified the trajectory assertions, it is the task of the theory to take over. The tools have verified that each individual abstract assertion holds for the circuit under the implementation mapping. The theory can make the claim that the entire abstract specification holds for the circuit under the implementation mapping. The user, however, has to ensure the double-sided restriction property on the abstract specification and implementation mapping.

Once individual operations have been verified, the methodology must be able to stitch operations together to reason about infinite execution sequences. As the first step in this direction, a closure construction of the main machine is used to define the effect of execution sequences on the pipeline structure.

This methodology has been used to verify parts of the Cobra-Lite processor. The Cobra-Lite processor is designed as a set of interconnected functional units. This thesis has concentrated on verifying one of these functional units, i.e., the fixed point unit (FXU), against the instruction set architecture of the processor. The abstract specification defines the instruction set architecture of the processor. The implementation mapping defines the environment around the FXU. The environment defines the set of protocols that is used to interface the FXU with the rest of the processor. In addition, the implementation mapping describes features such as system clocking, forwarding logic, instruction pipelines, and pipeline interlocks in the FXU. The thesis has presented results on our attempt to verify arithmetic and logical instructions in the FXU. Our approach seems to be the first one that can truly deal with the complexity of pipeline interlocks in modern processors.

The long-term objective is to verify the entire system, i.e., the Cobra-Lite processor. The current set of methodology and tools, however, cannot deal with the level of complexity of an entire processor. The initial focus is to verify each individual functional unit separately and then reason about the interactions among the functional units. Our work on the FXU indicates that this is a promising approach for formal verification of processors.

9.2 Future Work

Suggestions for future work can be divided in to the following categories: **1.** Languages **2.** Tools **3.** Theory **4.** Application **5.** Miscellaneous.

9.2.1 Languages

Future work in the languages area concentrates on the implementation mapping. Work in implementation mapping can be classified into the following categories: **1.** Extensions to the mapping to support complex processor features. **2.** Enable hierarchical verification. **3.** Explore alternative notations for describing the mapping.

Extensions: The current form of the implementation mapping has been used to describe features such as forwarding logic, pipeline interlocks, multiple cycle instructions, and multiple instruction issue in the FXU. The language, however, may require extensions to describe other features such as branch prediction, speculative execution, out-of-order execution, and register renaming.

The mapping has several forms to limit the scope of the cross-product construction. The main machine uses a nextmarker function to ensure that two instructions do not simultaneously occupy the same pipeline stage. The synchronization function synchronizes event vertices in the map machine to event vertices in the main machine. Map machines use the invalidate function to reduce nondeterminism in the trajectory assertion. It would be desirable to collapse all these forms into one single unified form.

Hierarchical verification: A large class of processors is designed as interconnected reactive subsystems. The subsystems have complex interfaces and use nondeterministic protocols to interact with each other. A clean and intuitive description of these interfaces requires a series of implementation mappings. Each level in the mapping serves to make the assertion more concrete. An abstract element at the highest level could be mapped into a complex protocol defined on several interface signals at the lowest level. The mapping language would have to be extended to describe such hierarchical mapping levels. This would form the basis for hierarchical verification, where a verification could be performed at each level of the mapping.

Alternative notations: One of the concerns in this work is that the implementation mapping can become very complex. An area of focus for future work would be to simplify or automate the generation of the mapping information as much as possible. Another possibility is to explore notations such as annotated timing diagrams for expressing the mapping. Formalisms on timing diagrams have been studied earlier[85][86]. Some of the concepts might be applicable to this problem.

9.2.2 Tools

Future work in the tools area can be classified broadly into two categories: **1.** Extensions for adaptive verification. **2.** Improving the efficiency and effectiveness of STE. Attempts at improving the efficiency and effectiveness of STE could be classified into: **2a.** A more efficient levelized version of STE. **2b.** Efficient memory modeling techniques. **2c.** Use of symmetry and abstraction **2d.** Issues of BDD variable ordering.

Adaptive verification: Section 5.3.2 classified set-based trajectory assertions into three categories in increasing order of expressiveness and complexity: 1) *Oblivious* 2) *Adaptive* and 3) *Prescient*. Consider a vertex v in the trajectory assertion. Now pick any two vertices v_i and v_j such that $(v, v_i) \in E$ and $(v, v_j) \in E$. An oblivious trajectory assertion was defined to have the restriction that $Set(\sigma_a(v_i)) \cap Set(\sigma_a(v_j)) = \emptyset$. An adaptive trajectory assertion was defined to have the restriction that either $Set(\sigma_a(v_i)) \cap Set(\sigma_a(v_j)) = \emptyset$ or $Set(\sigma_r(v_i)) \cap Set(\sigma_r(v_j)) = \emptyset$. The most general form without any such restrictions was called a prescient trajectory assertion. In Chapter 5, we restricted ourselves to oblivious trajectory assertions. It seems possible to extend set-based trajectory checking to adaptive trajectory assertions. This would require introducing the concept of universal and existential edges. As an example consider vertices v, v_1 and v_2 in an adaptive trajectory assertion such that $(v, v_1) \in E$ and $(v, v_2) \in E$. Let A_1 and R_1 be the set of action and reaction node assignments associated with vertex v_1, i.e. $A_1 = Set(\sigma_a(v_1))$ and $R_1 = Set(\sigma_r(v_1))$. Similarly, let A_2 and R_2 be the set of action and reaction node assignments associated with vertex v_2. Since this is an adaptive trajectory assertion, we know that either $A_1 \cap A_2 = \emptyset$ or $R_1 \cap R_2 = \emptyset$. If $A_1 \cap A_2 = \emptyset$, keep the graph as is. However, if $R_1 \cap R_2 = \emptyset$, modify the graph as show in Figure 9.1. The edges (u_{ab}, v_a) and (u_{ab}, v_b) represent a set of existential edges, i.e., either of the edges is acceptable.

The three-way branch in the modified graph can be interpreted as follows: 1. If the next stimulus belongs to the set $A_1 \cap A_2$, then the next circuit response can belong to the set R_1 or the set R_2. 2. If the next stimulus belongs to the set $A_1 \cap \overline{A_2}$, then the next circuit response has to belong to the set R_1. 3. If the next stimulus belongs to the set $\overline{A_1} \cap A_2$, then the next circuit response has to belong to the set R_2. Now the next stimulus and next circuit response together will define a unique vertex in the trajectory assertion.

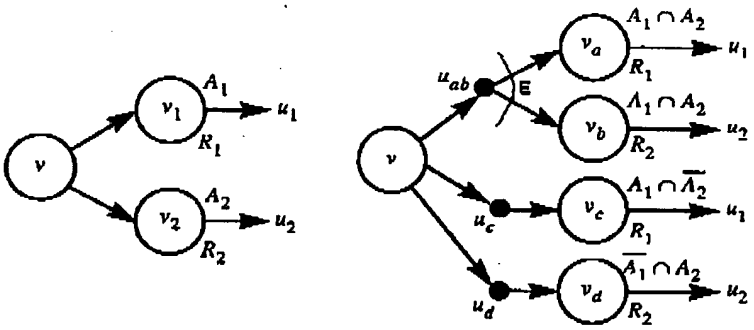


Figure 9.1: Graph modification when $R_1 \cap R_2 = \varnothing$.

It is possible to generalize the above modification for an arbitrary number of vertices and write a set-based trajectory checking algorithm. The problem is that there does not seem to be a corresponding lattice-based trajectory checking algorithm. The reason is that the complement operator on sets does not have a corresponding operator in the lattice-based formulation.

Levelized STE: Section 5.5.3 introduced the acyclic version of the STE algorithm. The algorithm computes the next state and updates the net values and Boolean functions for each state vertex in the trajectory assertion. Consider an acyclic trajectory assertion with n state vertices. The algorithm requires n next-state computations and n updates. It is possible to reduce the number of next-state computations by traversing the trajectory graph in a levelized order. A levelized version of the acyclic STE algorithm is shown in Figure 9.2. The function $len(w)$ represents the length of the longest path from the source to the vertex w. The levelized algorithm performs only $len(t) - 1$ next-state computations, where t is the sink of the trajectory assertion.

```
STE_Acyclic(G) {
  for (i = 1 to len(t)-1) {
    Ỹ = Sim(Q̃)
    foreach state vertex v so that len(v) = i {
      Q̃ = Path(v) ? Ỹ ⊔ Lat(σ_a(v)) : Q̃
      OK_A = OK_A · [¬Path(v) + [Ỹ ~ Lat(σ_a(v))]]
      OK_R = OK_R · [¬Path(v) + [Q̃ ⊒ Lat(σ_r(v))]]
    }
  }
}
```

Figure 9.2: A levelized acyclic STE algorithm.

The problem is that it is not apparent how to extend this levelized algorithm for generalized trajectory assertions.

Efficient memory modeling: As described in Section 1.4.6, Burch and Dill have used uninterpreted functions to compare a pipelined processor with a nonpipelined version of the processor. The use of uninterpreted functions allows Burch and Dill to specify the initial state of the entire memory by a single variable. It is possible to perform Burch and Dill style verification with BDDs. The problem is that the number of BDD variables required to specify the initial state of the memory would be prohibitively large. A variable would be required for each bit in the memory, so that the number of variables would be proportional to the size of the memory. It is, however, possible to replace the memory in the circuit by a behavioral model where the number of BDD variables would be proportional to the number of accesses to the memory[11]. The same technique can be incorporated into STE, enabling the verification of circuits with large embedded memories.

Symmetry and Abstraction: Verification of actual circuit designs can be expensive both in terms of CPU time and memory usage. It would be useful to use some form of symmetry or abstraction to reduce the complexity of the verification task. Researchers have used both structural and data symmetries to verify large static RAM circuits with Symbolic Trajectory Evaluation[24]. As an example, similar techniques could be used to reduce the verification to a single bit for bit-wise operations on words.

BDD variable ordering: Currently our tools use a simple minded user-defined variable ordering. Future work would involve exploring various automated techniques such as dynamic variable reordering[71]. Sifting has emerged as one of the most successful algorithms for dynamic variable ordering. Recently researchers have looked at an extension of sifting called group sifting[74]. Group sifting groups variables based on their affinity to their neighbors and then sifts groups of variables simultaneously. Symbolic Trajectory Evaluation would probably benefit most from some form of group sifting. The trajectory generation phase could provide helpful hints on how to group variables.

9.2.3 Theory

Future work in theory can be classified into the following categories: 1. Exploring the spectrum between model checking and STE. 2. Relation between trajectory assertion and temporal logic. 3. Completeness aspect of our methodology.

Model checking and Symbolic Trajectory Evaluation: Chapter 5 has presented a range of verification algorithms from set-based trajectory checking to lattice-based Symbolic Trajectory Evaluation. The set-based trajectory checking algorithm has the flavor of some model checking algorithms. One area of future work is to adopt and incorporate more model checking style algorithms so as to enable verification of a wider range of temporal behavior. Model checking style algorithms would enable verification of parts of the processor that operate as concurrent finite-state machines communicating via synchronous protocols. In fact one can envision an entire spectrum of algorithms and representation. At one end of the spectrum is the fast but sometimes overly pessimistic lattice-based representation and algorithms that can be used to verify a limited set of temporal behavior. At the other end is a more communication style paradigm which is more precise and can be used to model a wider range of temporal behavior but is computationally expensive.

Trajectory assertions and temporal logic: It would be interesting to compare the expressiveness of our trajectory assertions with various forms of temporal logic. Linear-time temporal logic (LTTL) assumes implicit universal quantification over all paths[68]. Branching-time temporal logic (BTTL) such as CTL allows explicit existential and universal quantification over paths[30].

Our oblivious trajectory assertions assume universal quantification over all paths. Oblivious trajectory assertions seem to be related in some form to LTTL. Alur and others have recently introduced an alternating-time temporal logic (ATTL)[69]. ATTL offers explicit existential and universal quantification over selected paths. These selected paths are viewed as possible outcomes of a two-player game in which the circuit and environment are the two players and they alternate moves. Our adaptive trajectory assertions have a similar sort of flavor. The universally quantified edges represent moves made by the environment. The existentially quantified edges represent moves made by the circuit. The difference, however, is that in our trajectory assertions the circuit and the environment don't have to strictly alternate moves.

Completeness: Beatty reduced the task of verification for all possible execution sequences into a check for three separate properties, i.e., *Obedience*, *Conformity*, and *Distinction*[13][15]. The Obedience property is the check to verify that the trajectory assertion holds for the circuit. The Conformity property requires that for every specification input sequence, there should be a corresponding circuit input sequence. The Distinction property requires that two distinct specification output sequences should have distinct circuit output sequences. Note the fact that Conformity and Distinction are properties of the abstract specification and the implementation mapping. They do not require the circuit. This thesis has mainly dealt with the Obedience property. Chapter 7 used closure construction on the main machine to create infinite execution sequences. This, however, is incomplete. We need to perform some sort of closure construction on the set of trajectory assertions to ensure that the inputs and internal state elements are consistent or in other words conform with each other. Beatty was able to claim that the conformity check needs to be performed only on the abstract inputs. Internal state elements do not need to be checked for conformity since they appear on both sides of the assertion. It remains to be seen if Obedience, Conformity, and Distinction define the complete set of properties to verify all possible execution sequences in our extended framework.

9.2.4 Cobra-Lite Verification

Future work in Cobra-Lite verification falls into the following categories: 1. Verification of individual functional units. 2. Reason about interactions between functional units. 3. Methods to reduce complexity of verification. 4. Incorporation of new data structures in STE.

Individual functional units: Current work has concentrated on verifying the fixed point unit (FXU) in the Cobra-Lite processor. The next step is to use our methodology to verify the load store unit (LSU) and the branch processing unit (BPU). The LSU and BPU represent more complex functional units with a greater number of gates. The implementation mapping might have to be extended to describe the increasing levels of nondeterministic behavior in these functional units.

Interactions between functional units: Once the individual functional units have been verified, we need to reason about the interaction between the functional units. Consider the interaction between the FXU and LSU. The verification of the FXU required map machines that defined the interface protocols between the FXU and LSU. The verification of the LSU will use the mirror image of these map machines to describe the interface between the LSU and FXU. It remains to be seen how much work is required to reason about the interactions. Some model checking style algorithms might have to be incorporated into STE to reason about the interactions between the functional units.

Complexity of verification: An important area of future work would be to study various techniques to reduce the complexity of verification using STE. Some of the feasible techniques are as follows: 1. An efficient levelized version of STE. 2. Use of efficient memory models. 3. Use of symmetry and abstraction. 4. Better and automated variable ordering heuristics. 5. Automatic breaking of complex assertions into multiple simpler ones.

New data structures: Binary Decision Diagrams are sometimes unsuitable to efficiently represent the data path. Recently researchers have come up with alternative representations such as Binary Moment Diagrams (BMDs)[72] and Hybrid Decision Diagrams (HDDs)[73] that efficiently represent word-level functions. One way to deal with the complexity of the data path would be to incorporate these data structures into Symbolic Trajectory Evaluation.

9.2.5 Miscellaneous

This thesis has concentrated on using the abstract specification and implementation mapping for formal verification. These descriptions could also be used to perform *smart* or *directed simulation*. The trajectory assertion captures all possible nondeterministic cases that can arise in the circuit. The trajectory assertion could be used to intelligently pick a set of simulation patterns to exercise a larger part of the circuit, and provide some sort of a coverage metric to quantify what percentage of all possible nondeterministic cases had been covered by the simulation patterns.

Bibliography

Formal Verification

[1] P. Camurati and P. Prinetto, "Formal Verification of Hardware Correctness: Introduction and Survey of Current Research," *IEEE Computer, 21(7)*, pp. 8-19, July 1988.

[2] A. Gupta, "Formal Hardware Verification Methods: A Survey," *Formal Methods in System Design, 1(2/3)*, pp. 151-238, October 1992.

Simulation based Verification

[3] J. Darringer, "The Application of Program Verification Techniques to Hardware Verification." *16th Design Automation Conference*, pp. 375-381, 1979.

[4] D. S. Reeves and M. J. Irvin, "Fast Methods for Switch-Level Verification of MOS Circuits," *IEEE Transactions on CAD/IC, Vol. CAD-6, No. 5*, pp. 776-779, September 1987.

[5] S. Bose and A. L. Fisher, "Verifying Piplined Hardware Using Symbolic Logic Simulation," *International Conference on Computer Design*, pp. 217-221, 1989.

[6] S. Bose and A. L. Fisher, "Automatic Verification of Synchronous Circuits using Symbolic Logic Simulation and Temporal Logic," *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pp. 759-764, 1989.

[7] D. L. Beatty, R. E. Bryant, and C. J. H. Seger, "Synchronous Circuit Verification by Symbolic Simulation: An Illustration," *Sixth MIT Conference on Advanced Research in VLSI*, pp. 98-112, 1990.

[8] R. E. Bryant and C. J. H. Seger, "Formal Verification of Digital Circuits Using Symbolic Ternary System Models," *Computer-Aided Verification '90, American Mathematical Society*, pp. 121-146, 1991.

[9] R. E. Bryant, "Formal Verification of Memory Circuits by Switch-Level Simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 10, No. 1*, pp. 94-102, January 1991.

[10] R. E. Bryant, "A Methodology for Hardware Verification Based on Logic Simulation," *J. ACM, Vol. 38, No. 2*, pp. 299-328, April 1991.

[11] M. Velev, R. E. Bryant, and A. Jain, "Efficient Modeling of Memory Arrays in Symbolic Simulation," *To appear in Computer-Aided Verification, CAV-97*, June 1997.

Symbolic Trajectory Evaluation

[12] R. E. Bryant, D. L. Beatty, and C. J. H. Seger, "Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation," *28th Design Automation Conference*, pp. 397-402, June 1991.

[13] D. L. Beatty, "A Methodology for Formal Hardware Verification with Application to Microprocessors," *Ph.D. Thesis, published as technical report CMU-CS-93-190, School of Computer Science, Carnegie Mellon University*, August 1993.

[14] C. J. H. Seger, "Voss-A Formal Hardware Verification System: User's Guide," *Technical Report 93-45, Department of Computer Science, University of British Columbia*, 1993.

[15] D. L. Beatty and R. E. Bryant, "Formally Verifying a Microprocessor Using a Simulation Methodology," *31st Design Automation Conference*, pp. 596-602, June 1994.

[16] S. Hazelhurst and C. J. H. Seger, "Composing Symbolic Trajectory Evaluation Results," *Lecture Notes in Computer Science, Computer Aided Verification, 6th International Conference, CAV 94*, pp. 273-285, 1994.

[17] S. Hazelhurst and C. J. H. Seger, "A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and BDD's," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 14, No. 4*, pp. 413-422, April 1995.

[18] C. J. H. Seger and R. E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Formal Methods in System Design 6*, pp. 147-189, 1995.

[19] M. D. Aagaard and C. J. H. Seger, "The Formal Verification of a Pipelined Double-Precision IEEE Floating-Point Multiplier," *International Conference on Computer-Aided Design*, pp. 7-10, November 1995.

[20] M. Pandey, R. Raimi, D. L. Beatty, and R. E. Bryant, "Formal Verification of PowerPC™ Arrays Using Symbolic Trajectory Evaluation," *33rd Design Automation Conference*, pp. 649-654, June 1996.

[21] A. Jain, K. Nelson, and R. E. Bryant, "Verifying Nondeterministic Implementations of Deterministic Systems," *Lecture Notes in Computer Science, Formal Methods in Computer-Aided Design*, pp. 109-125, November 1996.

[22] M. Pandey, R. Raimi, R. E. Bryant, and M. S. Abadir, "Formal Verification of Content Addressable Memories Using Symbolic Trajectory Evaluation," *To appear in 34rd Design Automation Conference*, June 1997.

[23] K. Nelson, A. Jain, and R. E. Bryant, "Formal Verification of a Superscalar Execution Unit," *To appear in 34rd Design Automation Conference*, June 1997.

[24] M. Pandey and R. E. Bryant, "Exploiting Symmetry When Verifying Transistor-Level Circuits by Symbolic Trajectory Evaluation," *To appear in Computer-Aided Verification, CAV-97*, June 1997.

Language Containment

[25] L Gertner and R. P. Kurshan, "Logical Analysis of Digital Circuits," *Computer Hardware Description Languages and their Applications, Elsevier Science Publiser B. V., IFIP*, pp. 47-67 1987.

[26] R. P. Kurshan, "Reducibility in Analysis of Coordination," *Discrete Event Systems: Models and Applications, Lecture Notes in Control and Information Sciences*, pp. 19-39, 1987.

[27] R. P. Kurshan, "Analysis of Discrete Event Coordination," *Lecture Notes in Computer Science 430*, pp. 414-453, 1990.

[28] R. P. Kurshan, "Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach," *Princeton University Press*, 1994.

Model Checking

[29] E. M. Clarke and E. A. Emerson, "Synthesis of Synchronization Skeletons for Branching Time Temporal Logic," *Proceedings of the Workshop on Logic of Programs, Lecture Notes in Computer Science, Vol. 131*, May 1981.

[30] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specification," *ACM Transactions on Programming Languages and Systems, 8(2)*, pp. 244-263, 1986.

[31] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential Circuit Verification Using Symbolic Model Checking," *27th Design Automation Conference*, pp. 46-51, June 1990.

[32] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol Verification as a Hardware Design Aid," *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 522-525, 1992.

[33] K. L. McMillan, "Symbolic Model Checking," *Kluwer Academic Publishers*, 1993.

[34] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness, "Verification of the Futurebus+ cache coherence protocol," *Proceedings of the 11th International Symposium on Computer Hardware Description Languages and their Applications*, April 1993.

Theorem Provers

[35] M. J. C. Gordon, "HOL: A Proof Generating System for Higher-Order Logic," *VLSI Specification, Verification, and Synthesis, Kluwer Academic Press*, 1988.

[36] R. S. Boyer and J. S. Moore, "A Computational Logic Handbook," *Academic Press, Boston*, 1988.

[37] M. Bickford, C. Mills, and E. A. Schneider, "Clio: An Applicative Language-Based Verification System," *Technical Report 15-7, Odyssey Research Association, Ithaca, N.Y.*, March 1989.

[38] S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal Verification for Fault-tolerant Architectures: Prolegomena to the Design of PVS," *IEEE Transactions on Software Engineering, 21(2)*, pp. 107-125, February 1995.

[39] M. Kaufmann and J. S. Moore, "ACL2: An Industrial Strength Version of Nqthm," *Proceedings of the 11th Annual Conference on Computer Assurance, IEEE Computer Society Press*, pp. 23-34, June 1996.

Verification of Non-Pipelined Processors

[40] A. J. Cohn, "A Proof of Correctness of the Viper microprocessors: The First Level," *VLSI Specification, Verification and Synthesis, Kluwer*, pp. 27-72, 1988.

[41] D. Borrione, P. Camurati, J. L. Paillet, and P. Prinetto, "A Functional Approach to Formal Hardware Verification: The MTI Experience," *International Conference on Computer Design*, pp. 592-595, 1988.

[42] J. J. Joyce, "Multi-level Verification of Microprocessor-based Systems," *Ph.D. Thesis, published as technical report 195, University of Cambridge*, May 1990.

[43] W. A. Hunt Jr., "FM8501: A Verified Microprocessor," *Lecture Notes in Artificial Intelligence, Springer-Verlag*, 1994.

[44] P. J. Windley, "Formal Modeling and Verification of Microprocessors," *IEEE Transactions on Computers, 44(1)*, pp. 54-72, January 1995.

Verification of Pipelined Processors

[45] M. Srivas and M. Bickford, "Formal Verification of a Pipelined Microprocessor," *IEEE Software 7(5)*, pp. 52-64, September 1990.

[46] J. R. Burch and D. L. Dill, "Automatic Verification of Pipelined Microprocessor Control," *Lecture Notes in Computer Science, Computer Aided Verification, 6th International Conference, CAV 94*, pp. 68-80, 1994.

[47] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas, "Effective Theorem Proving for Hardware Verification," *Lecture Notes in Computer Science, Conference on Theorem Provers in Circuit Design*, pp. 203-222, 1994.

[48] P. J. Windley and M. Coe, "A Correctness Model for Pipelined Microprocessors," *Proceedings of the 1994 Conference on Theorem Provers in Circuit Design, Lecture Notes in Computer Science, Springer Verlag*, pp. 33-51, September 1994.

[49] P. J. Windley, "Verifying Pipelined Microprocessors," *IFIP Conference on Hardware Description Languages (CHDL 95), Tokyo, Japan*, August 1995.

[50] D. A. Appenzeller and A. Kuehlmann, "Formal Verification of a PowerPC™ Microprocessor," *International Conference on Computer Design, VLSI in Computers and Processors*, pp. 79-84, October 1995.

[51] R. B. Jones, D. L. Dill, and J. R. Burch, "Efficient Validity Checking for Processor Verification," *International Conference on Computer-Aided Design*, November 1995.

[52] M. K. Srivas and S. P. Miller, "Applying Formal Verification to the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods," *Formal Methods in System Design, 8(2)*, pp. 153-188, March 1996.

[53] J. R. Burch, "Technique for Verifying Superscalar Microprocessors," *33rd Design Automation Conference*, pp. 552-557, June 1996.

[54] P. J. Windley and J. R. Burch, "Mechanically Checking a Lemma Used in an Automatic Verification Tool," *Lecture Notes in Computer Science, Formal Methods in Computer-Aided Design*, pp. 362-376, November 1996.

[55] D. Cyrluk, "Inverting the Abstraction Mapping: A Methodology for Hardware Verification," *Lecture Notes in Computer Science, Formal Methods in Computer-Aided Design*, pp. 172-186, November 1996.

[56] B. Brock, M. Kaufmann, and J. S. Moore, "ACL2 Theorems about Commercial Microprocessors," *Lecture Notes in Computer Science, Formal Methods in Computer-Aided Design*, pp. 275-293, November 1996.

[57] J. Sawada and W. A. Hunt Jr., "Trace Table Based Approach for Pipelined Microprocessor Verification," *To appear in Computer-Aided Verification, CAV-97*, June 1997.

Documentation on Processors

[58] D. Best, C. Kress, N. Mykris, J. Russel, and W. Smith, "An Advanced-Architecture CMOS/SOS Microprocessor," *IEEE Micro*, pp. 11-26, August 1982.

[59] T. K. Miller III, B. L. Bhuva, R. L. Barnes, J.-C. Duh, H.-B. Lin, and D. E. Van den Bout, "The Hector Microprocessor," *International Conference on Computer Design*, pp. 406-411, 1986.

[60] "Cayuga Chip Specification: Release 2.2," *Technical Report, Computer Science and Electrical Engineering Departments, Cornell University, Ithaca, N.Y.*, December 1988.

[61] J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach," *Morgan Kaufmann Publishers*, 1990.

[62] K. W. Fernald, T. A. Cook, T. K. Miller III, and J. J. Paulos, "A microprocessor-based implantable telemetry system," *IEEE Computer, 24(3)*, pp. 23-30, March 1991.

[63] C. May, E. Silha, R. Simpson, and H. Warren, "The PowerPC Architecture: A Specification for a New Family of RISC Processors," *Morgan Kaufmann Publishers*, 1994.

[64] J. Hoskins and R. Dimmick, "Exploring the IBM AS/400 Advanced 36," *Maximum Press*, 1994.

[65] F. G. Soltis, "Inside the AS/400," *Duke Press*, 1996.

[66] S. Gilfeather, J. Gehman and C. Harrison, "Architecture of a Complex Arithmetic Processor for Communication Signal Processing," *SPIE Proceedings, International Symposium on Optics, Imaging and Instrumentation, 2296 Advanced Signal Processing: Algorithms, Architectures and Implementations V*, pp. 624-625, July 1994.

[67] T. Coe, "Inside the Pentium FDIV Bug," *Dr. Dobbs Journal, 20(4)*, pp. 129-135, April 1995.

Temporal Logic

[68] A. Pnueli, "The Temporal Logic of Programs," *Proceedings of the 3rd Symposium on Foundation of Computer Science*, pp. 46-57, 1977.

[69] R. Alur, T. A. Henzinger, and O. Kupferman, "Alternating-Time Temporal Logic," *Unpublished manuscript courtesy T. Henzinger*, December 1996.

Binary Decision Diagrams and Variations

[70] R. E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers, Vol. 35, No. 8*, pp. 677-691, August 1986.

[71] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *International Conference on Computer-Aided Design*, pp. 42-47, November 1993.

[72] R. E. Bryant and Y. A. Chen, "Verification of Arithmetic Circuits with Binary Moment Diagrams," *32nd Design Automation Conference*, pp. 535-541, June 1995.

[73] E. M. Clarke, M. Fujita, and X. Zhao, "Hybrid Decision Diagrams - Overcoming the Limitations of MTBDDs and BMDs," *International Conference on Computer-Aided Design*, pp. 159-163, November 1995.

[74] S. Panda and F. Somenzi, "Who Are the Variables in Your Neighborhood," *International Conference on Computer-Aided Design*, pp. 74-77, November 1995.

[75] Y. A. Chen and R. E. Bryant, "PBHD: An Efficient Functional Representation for Floating-Point Circuit Verification," *Unpublished manuscript, Submitted to International Conference on Computer-Aided Design*, 1997.

Hardware Description Languages

[76] M. R. Barbacci and D. P. Siewiorek, "Some Aspects of the Symbolic Manipulation of Computer Descriptions," *Technical Report, Carnegie Mellon University*, 74-25, July 1974.

[77] J. P. V. Tassel, "The Semantics of VHDL with VAL and HOL: Towards Practical Verification Tools," *Technical Report, University of Cambridge, CAMC 196*, June 1990.

[78] D. E. Thomas and P. Moorby, "The Verilog Hardware Description Language," *Kluwer Academic Publishers*, 1991.

[79] "UDL/I Language Reference," *UDL/I committee, Japan Electronic Industry Development Association*, Draft Version 1.0h4, 1991.

[80] Z. Navabi, "VHDL-Analysis and Modeling of Digital Systems," *Mc-Graw Hill Inc.*, 1993.

High-Level Synthesis

[81] R. Compasano and W. Rosenstiel, "Synthesizing Circuits from Behavioral Descriptions," *IEEE Transactions on Computer-Aided Design*, 8(2), pp. 171-180, February 1989.

[82] D. E. Thomas et al., "Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench," *Kluwer Academic Publishers*, 1990.

[83] Y. Nakamura, K. Oguri, and A. Nayoga, "Synthesis from Pure Behavioral Descriptions," *High-Level Synthesis*, pp. 205-229, 1991.

[84] W. Wolf et al., "The Princeton Behavioral Synthesis System," *29th Design Automatic Conference*, pp. 182-187, 1992.

Timing Diagrams

[85] G. Borriello, "A New Interface Specification Methodology and its Application to Transducer Synthesis," *Ph.D. Thesis, Computer Science Division, University of California at Berkeley*, 1988.

[86] G. Borriello, "Formalized Timing Diagrams," *European Design Automation Conference*, pp. 372-377, 1992.

Switch-Level Models

[87] R. E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Transactions on Computers, Vol. C-33, No. 2*, pp. 160-177, February 1984.

[88] R. E. Bryant, "Extraction of Gate Level Models from Transistor Circuits by Four-Valued Symbolic Analysis," *International Conference on Computer-Aided Design*, pp. 350-353, 1991.

Miscellaneous

[89] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "Introduction to Algorithms," *MIT Press, McGraw-Hill Book Company*, 1990.

Appendix A

Hardware Specification Language

NAME

HSL - format of the Hardware Specification Language

The Hardware Specification Language is used to describe the high-level behavior of a system. The Hardware Specification Language can be used to describe several interacting finite-state machines. Each finite-state machine is associated with abstract elements. Apart from these, the language allows the user to define global state elements that are visible to all machines. Each finite-state machine is defined as a set of assertions. The assertion defines a set of pre and post conditions that operate on state elements.

Multiple interacting finite-state machines are assumed to operate synchronously, i.e. machines make transitions in lock-step fashion. A future possible extension is to allow the user to define the concurrency model for multiple interacting machines.

DESCRIPTION

Reserved keywords for the Hardware Specification Language are given below.

AND	**ARRAY**	**BIT**	**DEFINE**
ENDMACHINE	**IS**	**LEADSTO**	**MACHINE**
OF	**RANGE**	**STATE**	**TO**
TYPE	**VARIABLE**	**WHEN**	

The lower case versions of these keywords are also reserved and can be used interchangeably.

LEXICAL CONVENTIONS

Vertical bar | indicates choice, Question Mark ? follows optional items, Star * follows items that can be repeated 0 or more times, Plus + follows items that can be repeated 1 or more times, and parentheses < > indicate grouping.

WhiteSpace ::= <*SpaceChar*>+

SpaceChar ::= *blank* | *tab* | *newline* | *carriage return*

Comment ::= *OneLineComment* | *BlockComment*

OneLineComment

// any text upto newline

BlockComment

/* any text string that doesn't contain "*/" */

Ident

Any sequence of letters, digits, dollar sign and underscore symbols, except keywords. The first character has to be a letter or underscore symbol. Idents are case sensitive.

IdentList ::= *Ident* <',' Ident>*

A list of identifiers separated by commas.

Integer

An integer.

Number

Any non-negative integer.

Value ::= *BinaryValue* | *OctalValue* | *HexValue*

Values are enclosed within double quotes. A prefix character can be used to specify a binary (B), octal (O) or hexadecimal (X) value, as shown below. If a prefix character is not specified, then the value is assumed to be a binary value.

BinaryValue ::= B?"<0|1>+"

In this notation, single bit values can be represented as B"0" or B"1". HSL permits '0' or '1' as alternative notations for bit values.

OctalValue ::= O"<0-7>+"

HexValue ::= X"<0-9|A-F>+"

ORGANIZATION

The Hardware Specification Language contains exactly one *Program*, as described by the following informal BNF. Terminal symbols are rendered in boldface and enclosed in single quotes. White space may be included between any items.

Program ::= <*TypeDefn*>* <*StateDecl*>* <*Machine*>*

HSL supports several interacting finite-state machines. Each finite-state machine has its own type definition and state declaration section. However, if the user requires certain global type definitions and state declarations to be visible to all machines, then they have to be defined before any machine descriptions.

TYPE DEFINITIONS

TypeDefn ::= 'TYPE' *Ident* 'IS' *Types* ';'

Type definitions give names to types. This makes it convenient to declare state and local variables by referring to type names.

Types

::= 'BIT'

|= 'RANGE' *Number* 'TO' *Number*

|= '(' *IdentList* ')'

|= *Ident*

|= 'ARRAY' '(' *Number* 'TO' *Number* ')' 'OF' *Types*

Basic types include bit, subranges of fixed non-negative integers, and enumerations. Types can also be identifiers corresponding to type definitions and fixed sized arrays

of these types. In arrays the first number refers to the most significant bit and the second number to the least significant bit. The ZLSB (zero is least significant bit) notation requires the first number to be greater than the second. The ZMSB (zero is most significant bit) notation requires the first number to be less than the second. Either notation can be used. However the same notation has to be used in all array declarations. The single bit type, integer range type, and single dimensional array of bit are collectively referred to as bitvector types. Enumerations and array of enumerations are referred to as enumerated types. Array of ranges and multidimensional arrays of bit are referred to as multidimensional types.

STATE DECLARATIONS

StateDecl ::= **'STATE'** *IdentList* **':'** *Types* **';'**

The state elements define the state space of the machine. The state elements can be classified into 3 categories based on their types, namely enumerated, bitvector and multidimensional states. Enumerated states are state elements of enumerated type. Bitvector states are state elements of bitvector type. The rest of the state elements are multidimensional states.

MACHINE DESCRIPTIONS

Machine ::=
'MACHINE' *Ident* **';'**
*<TypeDefn>**
*<StateDecl>**
*<AssertBlock>**
'ENDMACHINE'

Every machine has a unique name. Each machine can be associated with a set of type definitions and state element declarations that are visible only to this machine. The machine is defined as a set of assertions that operate on state elements.

AssertBlock ::=
Ident **'{'**
*<LocalDecl>**
<Assertion>
'}'

The assertion block has to be associated with a name. The name is used to identify the assertion in the implementation mapping and ordering descriptions. Assertions can be associated with a set of local variables. Note that the variables are local to the block and are not visible outside the assertion block.

LocalDecl ::= **'VARIABLE'** *IdentList* **':'** *Types* **';'**

Local variables can be classified into 3 categories based on their types, namely enumerated and bitvector variables. Enumerated variables are variables of enumerated type. Bitvector variables are variables of bitvector type. The rest of the local variables are multidimensional variables.

Assertion

::=

Clause <'AND' *Clause*\fR>*

'LEADSTO'

Clause <'AND' *Clause*>*

|=

'WHEN' '('*BoolExpr*')'

Clause 'AND' *Clause*>*

'LEADSTO'

Clause 'AND' *Clause*>*

The assertion defines a set of transitions in a finite-state machine. Each assertion is a pre and post-condition pair. The semantics of an assertion is that if the precondition holds at the current time, then after some passage of time the postcondition has to hold. However if the pre-condition does not hold at the current time, then the assertion places no restriction on the set of transitions. An assertion can be qualified with a Boolean expression. The **WHEN** qualifier defines a set of transitions only when the Boolean expression is true. The assertion places no restrictions on the set of transitions if Boolean expression is false. The qualifier has to be a Boolean expression on local variables.

Clause

::= '(' *State* 'IS' *Expr* ')'

|= '(' 'WHEN' '('*BoolExpr*')' *Clause* ')'

State ::= *Ident*<'['*Expr*']'>*

Each clause in the assertion defines a simple assignment or a conditional assignment to state elements. The right hand side expression is an expression over local variables. The Boolean expression in the conditional assignment is an expression over local variables. Enumerated states have to be assigned enumerations. Enumerated states corresponding to array of enumerations have to be indexed and assigned enumerations. Bitvector states can be assigned bitvector expressions. Bitvector states can be indexed and assigned bitvector expressions. Multidimensional states can be assigned multidimensional expressions. Multidimensional states can be also be indexed and assigned appropriate multidimensional or bitvector expressions. Indices have to be bitvector expressions.

EXPRESSIONS

Expr

::= *Integer*

|= *Value*

|= *Ident*<'['*Index*']'>*

|= '~' *Expr*

|= *Expr* '&' *Expr*

|= *Expr* '|' Expr

|= *Expr* '^' *Expr*

|= *Expr* '+' *Expr*

|= *Expr* '-' *Expr*

|= '<'*Expr* <','*Expr*>*'>'

There are 3 categories of expressions, namely, enumerated, bitvector, and multidimensional expressions. Enumerated expressions are restricted to enumerations. Bitvector expressions can be an integer, value, bitvector variables, and bitwise, arithmetic, and concatenation operations over these variables. Bitvector expressions can be signed or unsigned. Integers are assumed to be signed, values are unsigned, and all bitvector variables are unsigned. Based upon their arguments, the arithmetic operators perform the appropriate signed or unsigned addition and negation. Both the bitwise and arithmetic operators result in a signed expression if either of their arguments is signed. The concatenation is assumed to be signed if the leftmost expression is signed. Multidimensional expressions can be multidimensional variables and bitwise operations over these variables. The operator ~ is the bitwise negation operator. The operators and (&), xor (^), or (|) are the bitwise binary operators. The bitwise binary operators require the two arguments to be of the same size except if the arguments are bitvector expressions. Signed bitvector expressions are sign extended and unsigned bitvector are zero-extended. The operators + and - are the arithmetic operators. Concatenation is a set of comma separated bitvector expressions enclosed within < and >.

Index

::= *Expr*

|= *Expr* ':' *Expr*

Indices have to be bitvector expressions. An index can be a single expression or two colon seperated expressions denoting a range. In the ZLSB notation, the first expression should be greater than the second. In the ZMSB notation, the first expression should be less than the second.

BoolExpr

::= *BoolTerm*

|= '!' *BoolTerm*

|= *BoolTerm* '&&' *BoolTerm*

|= *BoolTerm* '||' *BoolTerm*

BoolTerm

::= '(' *BoolTerm* ')'

|= *Expr* '==' *Expr*

|= *Expr* '!=' *Expr*

|= *Expr* '>' *Expr*

|= *Expr* '<' *Expr*

|= *Expr* '>=' *Expr*

|= *Expr* '<=' *Expr*

The operators >, <, >=, <= are the relational operators. Relational operators operate on bitvector expressions. The relational operators can deal with both signed and unsigned bitvector expressions. The operators ==, != are the equality operators. Equality operators operate on both enumerated, bitvector, and multidimensional expressions.

The precedence and order of evaluation of the operators are the same as in the C programming language. The logical and bitwise negation operation have the highest precedence and associate from right to left. All other operators associate from the left to right. Next in order of decreasing precedence are the arithmetic operators followed by the relational operators followed by the equality operators. After that are the bitwise

operators with &, ^, | in decreasing precedence order. The logical operators are lowest in the precedence order with || having lower precedence than && .

EXAMPLE

Here is an example for a specification of an addressable accumulator. The specification can maintain the sum of signals for *m* different channels, storing the sum in its register array. There are two possible operations. The *store* operation stores the value on the data input in the addressed register. The *add* operation takes the sum of the data input and the addressed register and stores it back in the addressed register. In this specification, the register array has 4 registers with each register having 2 bits of data. The sizes have been defined with DEFINE statements that get expanded in the preprocessing stage in SPG.

```
/* 2 bits per word */
DEFINE DATASIZE 1
/* m = 4 words in the register file. */
DEFINE REGSIZE 3
/* Address Size = Log(REGSIZE) = 2 */
DEFINE ADDRSIZE 1
MACHINE accumulator;
    TYPE DATAWORD IS ARRAY (DATASIZE TO 0) OF BIT;
    TYPE ADDRWORD IS ARRAY (ADDRSIZE TO 0) OF BIT;
    TYPE OPS IS (store, add);

    STATE op: OPS;
    STATE address : ADDRWORD;
    STATE dataIn, dataOut : DATAWORD;
    STATE register : ARRAY (0 TO REGSIZE) OF DATAWORD;
    /*** Set Of Assertions ****/
    StoreAssertion {
        VARIABLE i : ADDRWORD;
        VARIABLE a : DATAWORD;
        (op IS store) AND (dataIn IS a) AND (address IS i)
            LEADSTO
        (dataOut IS a) AND (register[i] IS a)
    }

    AddAssertion {
        VARIABLE i : ADDRWORD;
        VARIABLE a, b : DATAWORD;
        (op IS add) AND (dataIn IS a) AND (address IS i) AND (register[i] IS b)
            LEADSTO
        (dataOut IS a+b) AND (register[i] IS a+b)
    }

    MaintainStateAssertion {
```

```
        VARIABLE i, j : ADDRWORD;
        VARIABLE b : DATAWORD;
        WHEN (i != j)
            (address IS i) AND (register[j] IS b)
                LEADSTO
            (register[j] IS b)
    }
ENDMACHINE
```

SOME FRAGILE EXTENSIONS OR SHORT FORMS

An expression of the form

'(' ExprA '<' ExprB '<' ExprC ')'

can be used as a short form for

'(' ExprA '<' ExprB ')' '&&' '(' ExprB '<' ExprC ')'

The expression

(FooVar[23:26] == B"0-01")

can be used as a short form for

(FooVar[23]==0) && (FooVar[25:26]==B"01")

Note that this is a fragile extension. This extension for constants can just be used for equality operators. For other operators, the behavior is undefined.

SEE ALSO

spg(1) ste(1) map(5) order(5)

D. L. Beatty, "A Methodology for Formal Hardware Verification, with Application to Microprocessors," *PhD Thesis, Carnegie Mellon University, CMU-CS-93-190, August '93*, pp. 78-96.

.BUGS AND LIMITATIONS

Should we allow the ability to define signed bitvector variables?

Have not defined the semantics when the state and expression in a clause have different sizes? A related question is what happens in an indexed expression if the index is out of range?

The problem with expression indexing is that do we implicitly assume index 0 corresponds to be the first element? How do we resolve this with array variables where the from field is non-zero? Also the other question should we allow range indexing for states?

In expressions, specifying a constant repeating pattern of the form:

|= Expr '#' Number

AUTHOR

Alok Jain, Carnegie Mellon University

THIS PAGE BLANK (USPTO)

Appendix B

Implementation Mapping Language

NAME

MAP - format for the implementation mapping.

The implementation mapping provides a mapping between the high-level specification and a low-level realization. The high-level specification is described in the Hardware Specification Language, **hsl**(5). The low-level realization can be described in the **vlog**(5) format.

The Hardware Specification Language consists of several interacting finite-state machine. Each machine is associated with a set of type definitions, state variables and a set of assertions. The implementation mapping defines a mapping for each of these machines The mapping has relevance only in the context of an HSL description. So read the hsl(5) man page before going through this man page.

DESCRIPTION

Reserved keywords for the mapping language are given below.

ALIAS	**AND**	**ARRAY**	**AT**
BIT	**CASE**	**COND**	**ELSE**
END	**ENDMACHINE**	**ENTITY**	**EVENT**
FROM	**INVALIDATE**	**IS**	**LEADSTO**
MACHINE	**MAP**	**NEXT**	**NEXTMARKER**
NODE	**OF**	**PHASE**	**RANGE**
STATE	**SYNCH**	**TO**	**TYPE**
VARIABLE	**VERTEX**	**WHEN**	

The lower case versions of these keywords are also reserved and can be used interchangeably.

LEXICAL CONVENTIONS

Vertical bar | indicates choice, Question Mark ? follows optional items, Star * follows items that can be repeated 0 or more times, Plus + follows items that can be repeated 1 or more times, and parentheses < > indicate grouping.

WhiteSpace ::= <*SpaceChar*>+

SpaceChar ::= *blank* | *tab* | *newline* | *carriage return*

Comment ::= *OneLineComment* | *BlockComment*

OneLineComment

// any text upto newline

BlockComment

/* any text string that doesn't contain "*/" */

Ident

Any sequence of letters, digits, dollar signs, and underscore symbols, except keywords. The first character has to be a letter or underscore symbol. Idents are case sensitive.

IdentList ::= *Ident* <',' Ident>*

A list of identifiers separated by commas.

EscapedIdent ::= *Ident* <',' *Ident*>*

Escaped Indentifiers start with the backslash character and can include any printable ASCII character. Escaped Identifiers end with white space.

Name ::= *Ident* | *EscapedIdent*

Names are case sensitive.

Integer

An integer.

Number

Any non-negative integer.

Value ::= *BinaryValue* | *OctalValue* | *HexValue*

Values are enclosed within double quotes. A prefix character can be used to specify a binary (B), octal (O), or hexadecimal (X) value, as shown below. If a prefix character is not specified, then the value is assumed to be a binary value.

BinaryValue ::= B?"<01>+"

In this notation, single bit values can be represented as B"0" or B"1". HSL permits '0' or '1' as alternative notations for bit values.

OctalValue ::= O"<0-7>+"

HexValue ::= X"<0-9|A-F>+"

ORGANIZATION

The mapping language contains exactly one *Program*, as described by the following informal BNF. Terminal symbols are rendered in boldface and enclosed in single quotes. White space may be included between any items.

Program ::= *<Machine>**

HSL can describe several interacting finite-state machines. MAP provides a mapping for each of these machines.

Machine ::=

'MACHINE' *Ident* **';'**

*<TypeDefn>**

*<StateDecl>**

*<NodeDecl>**

*<AliasDefn>**

*<MappingBlock>**

'ENDMACHINE'

The machine name identifies a finite-state machine in the HSL description. The HSL description has a set of type definitions and state declarations associated with the machine. These type definitions and state declarations are assumed to be visible and can be used in defining the mapping for this machine. The type definition and state declaration section in the mapping can be used to declare an additional set of types and states. The node declarations declares a set of nodes for the machine. The alias definitions define a name mapping from the nodes to actual net names in the realization. The main component of the mapping is in the mapping blocks. The mapping blocks define a temporal and spatial mapping from the set of state elements (strictly assignment to state elements) to the set of nodes (again strictly assignment to nodes).

TYPE DEFINITIONS

TypeDefn ::= **'TYPE'** *Ident* **'IS'** *Types* **';'**

Type definitions give names to types. This makes it convenient to declare state and local variables by referring to type names.

Types

::= **'BIT'**

|= **'RANGE'** *Number* **'TO'** *Number*

|= **'('** *IdentList* **')'**

|= *Ident*

|= **'ARRAY'** **'('** *Number* **'TO'** *Number* **')'** **'OF'** *Types*

Basic types include bit, subranges of fixed non-negative integers, and enumerations. Types can also be identifiers corresponding to type definitions and fixed sized arrays of these types. In arrays the first number refers to the most significant bit and the second number to the least significant bit. The ZLSB (zero is least significant bit) notation requires the first number to be greater than the second. The ZMSB (zero is most significant bit) notation requires the first number to be less than the second. Either notation can be used. However the same notation has to be used in all array declarations. The single bit type, integer range type, and single dimensional array of bit are collectively referred to as bitvector types. Enumerations and array of enumerations are referred to as enumerated types. Array of ranges and multidimensional arrays of bit are referred to as multidimensional types.

STATE DECLARATIONS

StateDecl ::= **'STATE'** *IdentList* **':'** *Types* **';'**

The state elements define the state space of the machine. The state elements can be classified into 3 categories based on their types, namely enumerated, bitvector, and multidimensional states. Enumerated states are state elements of enumerated type.

Bitvector states are state elements of bitvector type. The rest of the state elements are multidimensional states.

NODE DECLARATIONS

NodeDecl ::= **'NODE'** *Node* <',' *Node*>* ';'

Node ::= *Ident* <'[' *Number* **'TO'** *Number* ']'>*

The node declaration section defines a set of nodes. A node can be a single node or an array of nodes. In the ZLSB (zero is least significant bit) notation, the first number should be greater than the second. In the ZMSB (zero is most significant bit) notation, the first number should be less than the second. A node either directly defines an actual net or can be aliased to a net in the realization. Array nodes have to be associated with an alias definition to map them into actual net names.

ALIAS DEFINITIONS

AliasDefn ::= **'ALIAS'** *NodeName ActualNetName* ';'

NodeName ::= *Ident*<'['*Ident*']'>*

ActualNetName ::= < '{'*Name*'}'<'['*Expr*']'>*>+

The alias definition can be used to map nodes into actual net names in the realization. Array nodes have to be associated with an alias definition. The node name identifies a node from the set of node declarations. Array nodes have to be associated with a set of distinct variables corresponding to node indices. These are implicit index variables whose size is picked automatically from the node declarations. The actual net name is a concatenation of names and expression indices. Expression indices are bitvector expressions over the implicit variables introduced in the node names. The implicit variables are local to the alias definition.

As an example, assume the following node declaration and corresponding alias definition:

NODE ram[1 TO 2][0 TO 2];

ALIAS ram[i][j] {ram.}[i]{.}[2-j];

In this example *i* and *j* are implicit variables. The nodes ram[1][0], ram[1][1], ram[1][2], ram[2][0], ram[2][1] and ram[2][2] are getting mapped into actual nets ram.1.2, ram.1.1, ram.1.0, ram.2.2, ram.2.1 and ram.2.0 respectively.

MAPPING BLOCKS

MappingBlock

::= *EntityBlock*

|= *MapBlock*

|= *AppendBlock*

|= *InvalidateBlock*

The mapping blocks provide a temporal and spatial mapping from the set of state elements (strictly assignment to state elements) to the set of nodes (again strictly assignment to nodes). The temporal aspect of the mapping alongwith the flow of control is defined in terms of "control graphs." The entities define a hierarchy of control graphs which defines the temporal hierarchy for the realization. The upper most entity in the hierarchy (which we shall refer to as the "main" entity) defines the flow of control for the assertions in the HSL. The map blocks define a mapping for an assignment to each

state element in the machine. The temporal aspect of the mapping is captured by a control graph in the map block which is related to the "main" control graph. The spatial aspect is captured by a set of labellings on the control graph. The append block is used to add clauses to the assertion, which are specific for this realization. And finally the invalidate block is used to invalidate certain compositions of the control graphs.

CONTROL GRAPH

VertexDecl ::= **'VERTEX'** *IdentList* **':'** *VertexTypes* **';'**

VertexTypes

::= **'EVENT'**

|= **'PHASE'**

|= *Ident*

The control graph has two type of vertices. The event vertices represent instantaneous time points. The state vertices represent durations of time. The basic state vertex type is a phase-level vertex. The state vertex type can also be an identifier corresponding to an entity. Entities have to be defined before they can be used as vertex types.

NextDefn ::=

'NEXT' **'{'**

Edges

'}'

Edges

::= *Ident* **':'** *Ident* **';'**

|= *Ident* **':'** **'{'** *IdentList* **'}'** **';'**

The next definition defines the edges in the control graph. Identifiers correspond to vertices declared in the vertex declaration section for this control graph. All vertices (except the sink vertex) should appear once and only once on the left hand side. A single edge can be defined from a vertex on the left-hand side to a vertex on the right hand side. Or a set of edges can be described from a single vertex to a set of vertices. The control graph should have a single unique vertex with no incoming edges. This vertex serves as the source. Also, the control graph should have a single unique vertex with no outgoing edges. This vertex serves as the sink.

ENTITY BLOCKS

EntityBlock ::=

'ENTITY' *Ident* **'{'**

*<VertexDecl>**

NextDefn

<NextMarkerFunction>?

MapClauses

'}'

Entities are used to describe a hierarchy of control graphs. The vertex declarations and the next definition define the control graph associated with the entity. Each machine should have one (and only one) entity associated with the nextmarker function. The entity with the nextmarker function serves as the "main" entity. The map clauses define a labelling on the state vertices in the control graph.

NextMarkerFunction ::=

'NEXTMARKER' '{'

Ident ':' *MainEventVertex* ';'

'}'

The NextMarker function is defined on event vertices. The identifier on the left-hand side identifies an event vertex in this control graph. The main vertex is an event vertex in the "main" control graph.

MainEventVertex ::= *Ident*'.'*Ident*

The first identifier corresponds to the name of the "main" entity. The second identifier identifies an event vertex in the "main" entity.

MAP BLOCKS

MapBlock

::=

'MAP' '('*StateClause*')' **'TO'** '{'

*<VertexDecl>**

NextDefn

SynchFunction

*<LocalDecl>**

MapClauses

'}'

||=

'MAP' '('*StateClause*')' **'TO'** '{'

*<VertexDecl>**

NextDefn

SynchFunction

*<LocalDecl>**

'COND' *CondClause* < **'AND'** *CondClause* >* '{'

MapClauses

'}'

'}'

The map block provides a mapping for a state clause. A state clause corresponds to an assignment to a state element. The vertex declarations and the next definition define the control graph associated with the map block. The synch function synchs the control graph to the "main" control graph. The map block can be associated with a set of local variables that are used in the map clauses. The map clauses defines a labelling on the state vertices in the control graph. The mapping for this state clause may be dependent on other state clauses in the HSL. The cond clauses can be used to define state-dependent mappings.

StateClause ::= *StateTerm* **'IS'** *Ident*

The stateterm identifies a unique state element in the HSL description. The identifier is a distinct variable. This is an implicit variable whose type is picked automatically from the state declaration in HSL.

StateTerm ::= *Ident* <'['*Ident*']'>*

The stateterm is identifying a unique state in the HSL description. Array states can be associated with a set of distinct variables corresponding to array indices. These are implicit index variables whose type is picked automatically from the state declaration in HSL. The indexing level in the stateterm has to be equal to or deeper than the deepest indexing level used in all the assertions for this machine.

CondClause

::= '('*StateClause*')'

|= '('*StateClause* **'SYNCH'** *MainEventVertex*')'

The cond clause is used to define state-dependent mappings. It is used to bring in information about other states in the assertion. The cond clause is either a simple state clause or a state clause with synch information. The main vertex identifies an event vertex in the "main" control graph. And this synch information overrides the synch information in the map block for the state clause.

SynchFunction ::=

'SYNCH' '{'

Ident ':' *MainEventVertex* ';'

'}'

The synch function is defined on event vertices. The identifier on the left-hand side identifies an event vertex in this control graph. The main vertex is an event vertex in the "main" control graph.

APPEND ASSERTION BLOCK

AppendBlock ::=

'APPEND' *Ident* '{'

*<LocalDecl>**

Clause <**'AND'** *Clause*>*

'LEADSTO'

Clause <**'AND'** *Clause*>*

'}'

The identifier in the append block identifies an assertion block in the HSL. All local variables in the HSL assertion block are visible and can be used within this block. Additional local variables can be declared. The clauses are appended to the HSL assertion block.

Clause

::= '(' ')'

|= '(' *State* **'IS'** *Expr* ')'

|= '(' **'WHEN'** '('*BoolExpr*')' *Expr* ')'

|= '(' *State* **'IS'** *Expr* **'SYNCH'** *MainEventVertex* ')'

|= '(' **'WHEN'** '('*BoolExpr*')' *Expr* **'SYNCH'** *MainEventVertex* ')'

State ::= *Ident*<'['*Expr*']'>*

Each clause in the append assertion is either empty, defines a simple assignment or a conditional assignment to state elements. The difference between clauses in HSL and MAP is that the clauses in the mapping can be associated with synch information. The

synch information overrides the synch information in the map block for the state element.

INVALIDATE BLOCKS

InvalidateBlock ::=

'INVALIDATE' '{'

< '(' *MapStateVertex* <',' *MapStateVertex*>+ ')' ';' >+

'}'

Restrict the composition so as to invalidate certain combination of state vertices in map blocks.

MapStateVertex ::= *Ident*'.'*Ident*

The first identifier corresponds to a state element. The second identifier identifies a state vertex in the map block for the state element.

MAP CLAUSES

MapClauses ::= *MapClause* < **'AND'** *MapClause* >*

MapClause

::= *LabelClause*

|= *CaseClause*

|= *WhenClause*

The map clause can be a leaf-level label, a case statement, or a when statement.

LabelClause ::=

'LABEL' '{'

Label < **'AND'** *Label* >*

'}'

Label

::= '(' *Label* ')'

|= *Ident* <'['*Expr*']'>* **'IS'** *Expr* **'AT'** *PhaseVertex*

|= *Ident* <'['*Expr*']'>* **'IS'** *Expr* **'FROM'** *PhaseVertex* **'TO'** *PhaseVertex*

The label clause defines assigns expressions to nodes at particular time points in the control graph. The identifier is a node defined in the node declaration section. Expressions are over the implicit and local variables defined in this block. Indices have to be bitvector expressions. The phase vertex is the full path name for a phase-level vertex in the control graph hierarchy.

PhaseVertex ::= *Ident*<'.'*Ident*>*

The phase vertex identifies a specific instance of a phase-level vertex. The first identifier is a state vertex in the control graph for this mapping. If the state vertex type is an entity, then the second identifier is a state vertex in the entity control graph. And so on.

WhenClause

::=

'WHEN' **'('** *BoolExpr* **')'** **'{'**

MapClauses

'}'

||=

'WHEN' **'('** *BoolExpr* **')'** **'{'**

MapClauses

'}' **'ELSE'** **'{'**

MapClauses

'}'

Clauses can be associated with a Boolean restriction. The Boolean expression is over local and implicit variables in this block.

CaseClause ::=

'CASE' **'('** *Ident* **')'** **'{'**

<**'IS'** *Expr* **':'** *MapClauses* **';'**>+

'}'

The case statement performs a multi-way branch on the variable based upon the expressions. The case identifier is one of the implicit variables in this block. The case variable is limited to be an enumerated or bitvector variable. For an enumerated case variable, the expressions are limited to be simple enumerations. For a bitvector case variable, the expressions are limited to be bitvector constants.

LOCAL VARIABLES IN BLOCKS

LocalDecl ::= **'VARIABLE'** *IdentList* **':'** *Types* **';'**

Both alias and map blocks can be associated with local variables. The types syntax is the same as in HSL and has been reproduced below.

Types

::= **'BIT'**

||= **'RANGE'** *Number* **'TO'** *Number*

||= **'('** *IdentList* **')'**

||= *Ident*

||= **'ARRAY'** **'('** *Number* **'TO'** *Number* **')'** **'OF'** *Types*

Basic types include bit, subranges of fixed non-negative integers and enumerations. Types can also be identifiers corresponding to type definitions and fixed sized arrays of these types. In arrays the first number refers to the most significant bit and the second number to the least significant bit. In the ZLSB notation, the first number should be greater than the second. In the ZMSB notation, the first number should be less than the second. Identifiers correspond to visible type definitions in HSL. The single bit type, integer range type, and single dimensional array of bits are collectively referred to as bitvector types. Enumerations and array of enumerations are referred to as enumerated types. Array of ranges and multidimensional array of bits are referred to as multidimensional types.

EXPRESSIONS

The expression syntax and semantics is the same as in HSL. For the sake of completeness, it has been reproduced below.

Expr

::= *Integer*

|= *Value*

|= *Ident*<'['*Index*']'>*

|= '~' *Expr*

|= *Expr* '&' *Expr*

|= *Expr* '|' Expr

|= *Expr* '^' *Expr*

|= *Expr* '+' *Expr*

|= *Expr* '-' *Expr*

|= '<'*Expr* <','*Expr*>*'>'

There are 3 categories of expressions, namely enumerated, bitvector, and multidimensional expressions Enumerated expressions are restricted to enumerations. Bitvector expressions can be an integer, value, bitvector variables, and bitwise, arithmetic, and concatenation operations over these variables. Bitvector expressions can be signed or unsigned. Integers are assumed to be signed, values are unsigned, and all bitvector variables are unsigned. Based upon their arguments, the arithmetic operators perform the appropriate signed or unsigned addition and negation. Both the bitwise and arithmetic operators result in a signed expression if either of their arguments is signed. The concatenation is assumed to be signed if the left most expression is signed. Multidimensional expressions can be multidimensional variables and bitwise operations over these variables. The operator ~ is the bitwise negation operator. The operators and (&), xor (^), or (|) are the bitwise binary operators. The bitwise binary operators require the two arguments to be of the same size except if the arguments are bitvector expressions. Signed bitvector expressions are sign extended and unsigned bitvector are zero-extended. The operators + and - are the arithmetic operators. Concatenation is a set of comma separated bitvector expressions enclosed within < and >.

Index

::= *Expr*

|= *Expr* ':' *Expr*

Indices have to be bitvector expressions. An index can be a single expression or two colon seperated expressions denoting a range. In the ZLSB notation, the first expression should be greater than the second. In the ZMSB notation, the first expression should be less than the second.

BoolExpr

::= *BoolTerm*

|= '!' *BoolTerm*

|= *BoolTerm* '&&' *BoolTerm*

|= *BoolTerm* '||' *BoolTerm*

BoolTerm

::= '('*BoolTerm*')'

||= *Expr* '==' *Expr*

||= *Expr* '!=' *Expr*

||= *Expr* '>' *Expr*

||= *Expr* '<' *Expr*

||= *Expr* '>=' *Expr*

||= *Expr* '<=' *Expr*

The operators >, <, >=, <= are the relational operators. Relational operators operate on bitvector expressions. The relational operators can deal with both signed and unsigned bitvector expressions. The operators ==, != are the equality operators. Equality operators operate on both enumerated, bitvector, and multidimensional expressions.

The precedence and order of evaluation of the operators are the same as in the C programming language. The logical and bitwise negation operation have the highest precedence and associate from right to left. All other operators associate from the left to right. Next in order of decreasing precedence are the arithmetic operators followed by the relational operators followed by the equality operators. After that are the bitwise operators with &, ^, | in decreasing precedence order. The logical operators are lowest in the precedence order with || having lower precedence than && .

EXAMPLE

Here is an example of a mapping for a pipelined addressable accumulator. The corresponding specification is in the example section in hsl(5). The mapping is for a pipelined addressable accumulator circuit where the register access and adder operation occur simultaneously. A hold register has been incorporated between the adder and register array. Bypass logic handles the case where the same address is used on consecutive operations. The mapping first defines a set of node declarations. Note that the DEFINE statements in the HSL are visible in the mapping and can be used to define the size of the nodes. These will be expanded by a preprocessor in spg. The node declaration section is followed by a set of alias definitions which define how the node names are mapped into actual net names in vlog. An entity is used to define the cycle-level vertex. The cycle consists of 4 non-overlapping phases. The main entity defines flow of control. The flow of control has 3 cycles corresponding to the previous, current, and next instruction. The append block is used to define the address in the previous and next cycles. All the map blocks have a control graph with a single state vertex. These control graphs are synched with the current cycle in the main control graph. The mapping for *dataIn*, *address*, and *dataOut* are simple leaf-level mappings. The mapping for *op* uses a case statement. The mapping for the register array is a complex one that defines the pipelining in the circuit. If the currently addressed abstract register is the same as the address in the previous cycle, then the effective data is in the hold register. Else the effective data is in the register array.

```
MACHINE accumulator;
    NODE phi1, phi2, Clear;
    NODE addr[ADDRSIZE TO 0];
    NODE regL[REGSIZE TO 0][DATASIZE TO 0];
    NODE regH[REGSIZE TO 0][DATASIZE TO 0];
    NODE in[DATASIZE TO 0], out[DATASIZE TO 0], hold[DATASIZE TO 0];
```

```
ALIAS addr[x] {\Addr. }[x];
ALIAS in[x] {\In. }[x];
ALIAS out[x] {\Out. }[x];
ALIAS hold[x] {\D$Hold-L. }[x];
ALIAS regL[x][y] {\D$Reg.L. }[x]{.}[y];
ALIAS regH[x][y] {\D$Reg.H. }[x]{.}[y];

ENTITY cycle {
    VERTEX ps1, ps2, ps3, ps4: PHASE;
    VERTEX pe1, pe2, pe3, pe4, pe5: EVENT;
    NEXT {
        pe1: ps1;
        ps1: pe2;
        pe2: ps2;
        ps2: pe3;
        pe3: ps3;
        ps3: pe4;
        pe4: ps4;
        ps4: pe5;
    }
    LABEL {
        (phi1 IS '0' AT ps1) AND (phi2 IS '1' AT ps1) AND
        (phi1 IS '0' AT ps2) AND (phi2 IS '0' AT ps2) AND
        (phi1 IS '1' AT ps3) AND (phi2 IS '0' AT ps3) AND
        (phi1 IS '0' AT ps4) AND (phi2 IS '0' AT ps4)
    }
}
ENTITY main {
    VERTEX cs1, cs2, cs3: cycle;
    VERTEX ce1, ce2, ce3, ce4: EVENT;
    NEXT {
        ce1: cs1;
        cs1: ce2; /* Previous cycle */
        ce2: cs2;
        cs2: ce3; /* Current cycle */
        ce3: cs3;
        cs3: ce4; /* Next cycle */
    }
    NEXTMARKER {ce2: ce3;}
}
APPEND StoreAssertion {
    VARIABLE u : ADDRWORD;
    (address IS u SYNCH main.ce3) LEADSTO 0
}
```

```
APPEND AddAssertion {
    VARIABLE k, u : ADDRWORD;
    (address IS k SYNCH main.ce1) AND (address IS u SYNCH main.ce3)
        LEADSTO
    ()
}
APPEND MaintainStateAssertion {
    VARIABLE k, u : ADDRWORD;
    (address IS k SYNCH main.ce1) AND (address IS u SYNCH main.ce3)
        LEADSTO
    ()
}
MAP (op IS o) TO {
    VERTEX s1: cycle;
    VERTEX e1, e2: EVENT;
    NEXT { e1: s1; s1: e2; }
    SYNCH {e1: main.ce2;}
    CASE (o) {
        IS store: LABEL {Clear IS '1' AT s1.ps3}
        IS add: LABEL {Clear IS '0' AT s1.ps3}
    }
}
MAP (dataIn IS d) TO {
    VERTEX s1: cycle;
    VERTEX e1, e2: EVENT;
    NEXT { e1: s1; s1: e2; }
    SYNCH {e1: main.ce2;}
    LABEL {in IS d AT s1.ps3}
}
MAP (address IS v) TO {
    VERTEX s1: cycle;
    VERTEX e1, e2: EVENT;
    NEXT { e1: s1; s1: e2; }
    SYNCH {e1: main.ce2;}
    LABEL {addr IS v AT s1.ps3}
}
MAP (register[i] IS r) TO {
    VERTEX s1: cycle;
    VERTEX e1, e2: EVENT;
    NEXT { e1: s1; s1: e2; }
    SYNCH {e1: main.ce2;}
    COND (address IS k SYNCH main.ce1) {
        WHEN (i == k) { LABEL {hold IS ~r AT s1.ps2} }
        ELSE { LABEL { (regL[i] IS ~r AT s1.ps3) AND (regH[i] IS r AT s1.ps3) } }
```

```
        }
    }
    MAP (dataOut IS d) TO {|
        VERTEX s1: cycle;
        VERTEX e1, e2: EVENT;
        NEXT { e1: s1; s1: e2; }
        SYNCH {e1: main.ce2;}
        LABEL {out IS d AT s1.ps1}
    }
ENDMACHINE
```

SOME FRAGILE EXTENSIONS OR SHORT FORM

An expression of the form

'(' *ExprA* '<' *ExprB* '<' *ExprC* ')'

can be used as a short form for

'(' *ExprA* '<' *ExprB* ')' '&&' '(' *ExprB* '<' *ExprC* ')'

The expression

(FooVar[23:26] == B"0-01") .

can be used as a short form for .

(FooVar[23]==0) && (FooVar[25:26]==B"01")

Note that this is a fragile extension. This extension for constants can just be used for equality operators. For other operators, the behavior is undefined.

SEE ALSO

spg(1) ste(1) hsl(5) order(5) vlog(5)

D. L. Beatty, "A Methodology for Formal Hardware Verification, with Application to Microprocessors," *PhD Thesis, Carnegie Mellon University, CMU-CS-93-190, August '93*, pp. 163-174.

HACKS

All machine, state, enumeration, and node names within a module have to be distinct. Sometimes you might want a state name to be same as the node name. In that case you can use the alias mechanism as follows:

```
/* Assume state op is defined in HSL description for some machine. */
STATE op : BIT;
/* Now assume you want to define a node op for same machine */
NODE pseudo_op; // Define a node with some pseudo name.
ALIAS pseudo_op {op}; // Alias pseudo name into op|
```

BUGS

HSL allows only a single "main" entity. That implies that all assertions have to have a single flow of control. What do we do when we have multiple assertions with different flow of control?

I am not sure if the invalidate block should really be part of the mapping language. It seems to be a hack that can really be abused.

HSL clauses for which a mapping is not defined are assumed to have a NULL mapping. Maybe MAP should explicitly specify a NULL mapping.

Expression facility as in HSL is limited.

AUTHOR

Alok Jain, Carnegie Mellon University

X. Related Proceedings Appendix: Copies of Decisions Rendered by a Court or the Board in any Prior and Pending Appeals, Interferences or Judicial Proceedings

There are no related appeals or interferences to appellant's knowledge that would have a bearing on any decision of the Board of Patent Appeals and Interferences.